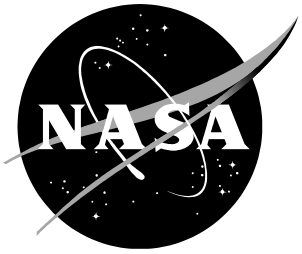
NASA/TM-20230013358



A Trajectory Algorithm to Support En Route and Terminal Area Self-Spacing Concepts: Fifth Revision

Terence S. Abbott
Langley Research Center, Hampton, Virginia

October 2023

NASA STI Program Report Series

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter-part of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

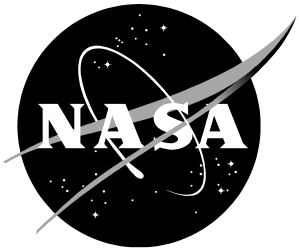
For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at http://www.sti.nasa.gov

- Help desk contact information:

https://www.sti.nasa.gov/sti-contact-form/ and select the "General" help request type.

NASA/TM-20230013358



A Trajectory Algorithm to Support En Route and Terminal Area Self-Spacing Concepts: Fifth Revision

Terence S. Abbott
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

October 2023

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Table of Contents

Nomenclature

2D:	2 dimensional
4D:	4 dimensional
ADS-B:	Automatic Dependence Surveillance Broadcast
BOD:	Bottom-Of-Descent
CAS:	Calibrated Airspeed
DTG:	Distance-To-Go
FAF:	Final Approach Fix
LDR:	Linear Deceleration Rate
MSL:	Mean Sea Level
RF:	Radius-to-Fix
STAR:	Standard Terminal Arrival Routes
TAS:	True Airspeed
TCP:	Trajectory Change Point
TOD:	Top-Of-Descent
TTG:	Time-To-Go
VTCP:	Vertical Trajectory Change Point

Subscripts

Subscripts associated with waypoints and TCPs, e.g., TCP_2, denote the location of the waypoint or TCP in the TCP list. Larger numbers denote locations closer to the end of the list, with the end of the list being the runway threshold. Subscripts in variables indicate that the variable is associated with the TCP with that subscript, e.g., $Altitude_2$ is the altitude value associated with TCP_2.

Units and Dimensions

Unless specifically defined otherwise, units (dimensions) are as follows:

time:	seconds
position:	degrees, + north and + east
altitude:	feet, above MSL
distance:	nautical miles
speed:	knots
track:	degrees, true, beginning at north, positive clockwise

Abstract

This document describes an algorithm for the generation of a four dimensional trajectory. Input data for this algorithm are similar to an augmented Standard Terminal Arrival (STAR) with the augmentation in the form of altitude or speed crossing restrictions at waypoints on the route. The algorithm calculates the altitude, speed, along path distance, and along path time for each of these waypoints. Wind data at each of these waypoints are also used for the calculation of ground speed and turn radius. This revision of the algorithm now accommodates linear decelerations between two speed-constrained waypoints. While this modification may appear trivial, the calculation of the deceleration rate cannot be accomplished using a closed-form solution. An iterative solution was developed that allowed for the variability of path distance due to speed influence on turn radii, Mach-CAS transition altitude, and the impact of wind on ground speed in calculating an accurate deceleration value.

Introduction

Concepts for self-spacing of aircraft operating into airport terminal areas have been under development since the 1970's (refs. 1-30). Interest in these concepts has recently been renewed due to a combination of emerging, enabling technology (Automatic Dependent Surveillance Broadcast data link, ADS-B) and the continued growth in air traffic with the ever increasing demand on airport (and runway) throughput. Terminal area self-spacing has the potential to provide an increase in the accuracy of runway threshold crossing times, which can lead to a decrease of the variability of the runway threshold crossing times. This decrease of the variability of the runway threshold crossing times can then lead to an increase in runway capacity through a reduction of the spacing buffers needed to assure safe separation during landing operations. Current concepts use a trajectory based technique that allows for the extension of self-spacing capabilities beyond the terminal area to a point prior to the top of the en route descent.

The overall NASA Langley concept for a trajectory-based solution for en route and terminal area self-spacing is fairly simple and is documented in references 31-33. By assuming a 4D trajectory for an aircraft and knowing that aircraft's position, it is possible to determine where that aircraft is on its trajectory. Knowing the position on the trajectory, the aircraft's estimated time-to-go (TTG) to a point can then be determined. To apply this to a self-spacing concept, a TTG is calculated for a leading aircraft and for the ownship. Note that the trajectories do not need to be the same. The nominal spacing time and spacing error can then be computed as:

nominal spacing time = planned spacing time interval + traffic TTG.

spacing error = ownship TTG – nominal spacing time.

The foundation of this spacing concept is the ability to generate a 4D trajectory. The algorithm presented in this paper uses as input a simple, augmented 2D path definition along with a forecast wind speed profile for each waypoint. This augmented 2D path definition would include horizontal waypoint information with relevant speed and altitude crossing constraints, with each speed or altitude constraint including the rate value required to meet the constraint. The algorithm then computes a full 4D trajectory defined by a series of trajectory change points (TCPs). The input speed (Mach or Calibrated Airspeed (CAS)) or altitude crossing constraint includes the deceleration rate or vertical angle value required to meet the constraint. The TCPs are computed such that speed values, Mach or CAS, and altitudes change linearly between them. TCPs also define the beginning and ending segments of turns, with the midpoint defined as a fly-by

waypoint. The algorithm also uses the waypoint forecast wind speed profile in a linear interpolation to calculate the wind speed at the altitude the computed trajectory crosses the waypoint. Wind speed values are then used to calculate the ground speeds along the path.

The major complexity in computing a 4D trajectory involves the interrelationship of ground speed with the path distance around turns. In a turn, the length of the estimated ground path and the associated turn radius will interact with the waypoint winds and with any change in the specified speed during the turn, i.e., a speed crossing-restriction at the waypoint. Either of these conditions will cause a change in the estimated turn radius. The change in the turn radius will affect the length of the ground path, which can then interact with the distance to the deceleration point, which thereby affects the turn radius calculation. To accommodate these interactions, the algorithm uses a multi-pass technique in generating the 4D path, with the ground path estimation from the previous calculation used as the starting condition for the current calculation. In a similar manner, this revision of the trajectory algorithm includes the ability to calculate the deceleration value to obtain a linear deceleration between two speed-constrained waypoints.

Algorithm Overview

The basic functions for this trajectory algorithm are shown in figure 1. Figure 1 also contains logic and some simple calculations that are not included in the body of this document, e.g., "restore the crossing angles." Also, note that waypoints are considered to be TCPs but not all TCPs are waypoints.

For the 2D input, the first and last waypoints must be fully constrained, i.e., have both a speed and altitude constraint defined. With the exception of the first waypoint, which is the waypoint farthest from the runway threshold, constraints must also include a variable that defines the means for meeting that constraint. For altitude constraints, this is the inertial descent angle; for speed constraints, it is the CAS deceleration rate. A separate, single Mach-to-CAS transition speed (CAS) value may also be input for profiles that involve a constant Mach / CAS descent segment. Additionally, an altitude / CAS restriction (e.g., in the U.S., the 10,000 ft / 250 kt restriction) may also be entered.

The algorithm computes the altitude and speed for each waypoint. It also calculates every point along the path where an altitude or speed transition occurs. These points are considered vertical TCPs (VTCPs). TCPs also define the beginning and ending segments of turns, with the midpoint defined as a fly-by waypoint. Turn data are generated by dividing the turn into two parts (from the beginning of the turn to the midpoint and from the midpoint to the end of the turn) to provide better ground speed (and resulting turn radius) data relative to a single segment estimation. A fixed, average bank angle value is used in the turn radius calculation. The algorithm also uses the forecast wind speed profile for a waypoint in a linear interpolation to calculate the wind speed at the altitude the computed trajectory crosses the waypoint (if the crossing altitude is not at a forecast altitude). For non-waypoint TCPs, the generator uses the forecast wind speed profile from the two waypoints on either side of the TCP in a double linear interpolation based on altitude and distance (to each waypoint). Of significant importance for the use of the data generated by this algorithm is that altitude and speeds (Mach or CAS) change linearly between the TCPs, thus allowing later calculations of DTG or TTG for any point on the path to be easily performed.

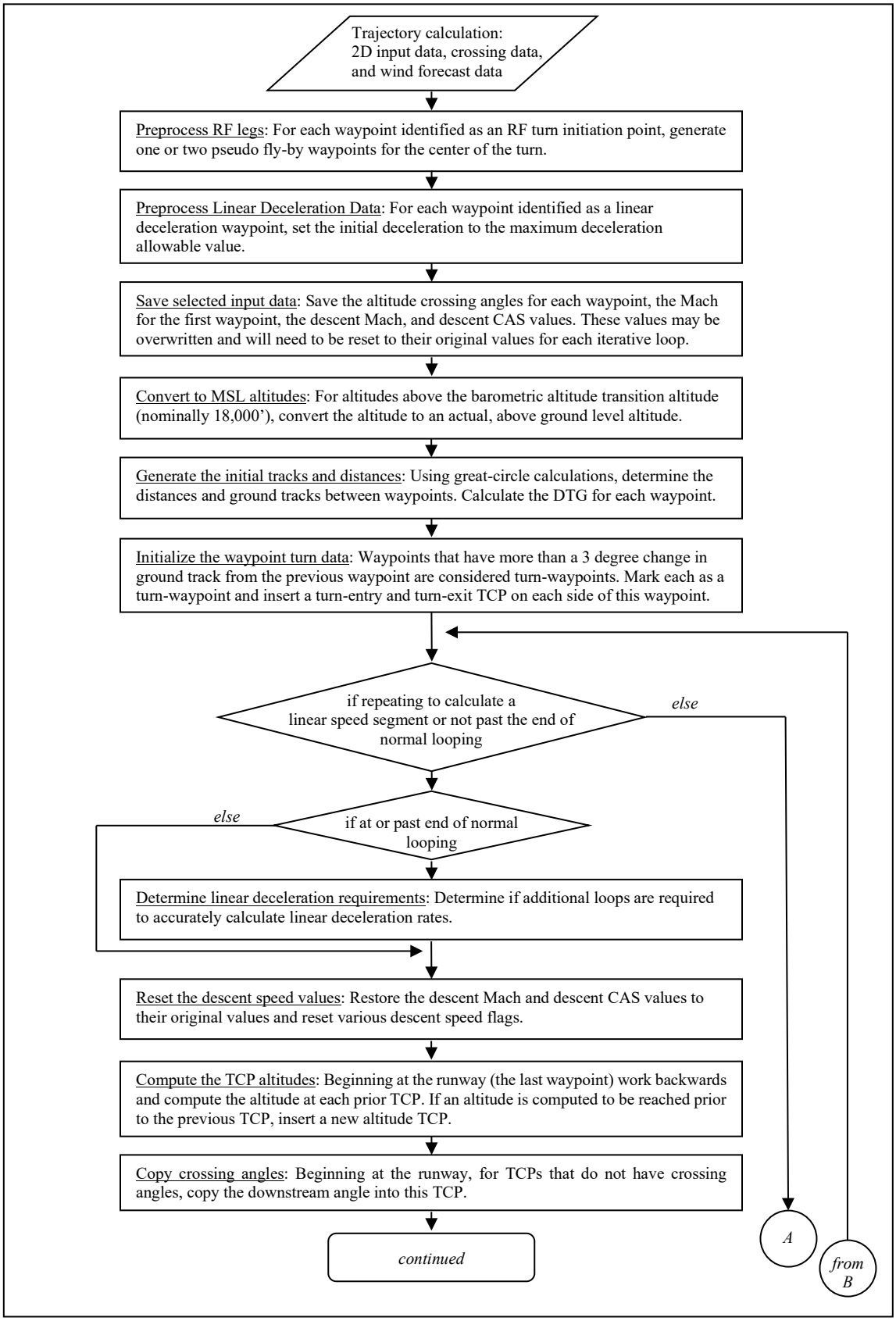


Figure 1. Basic functions.

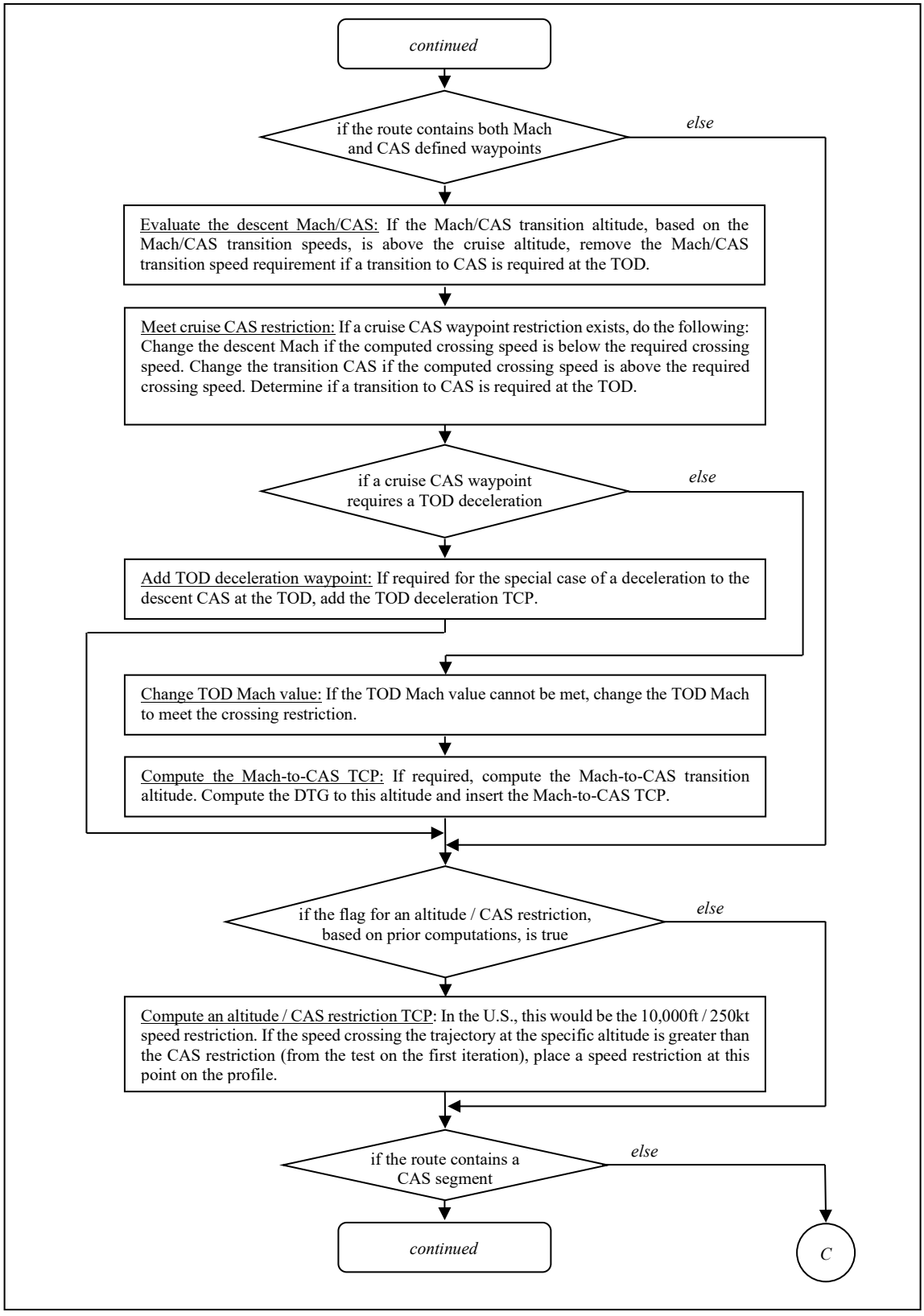


Figure 1 (continued). Basic functions.

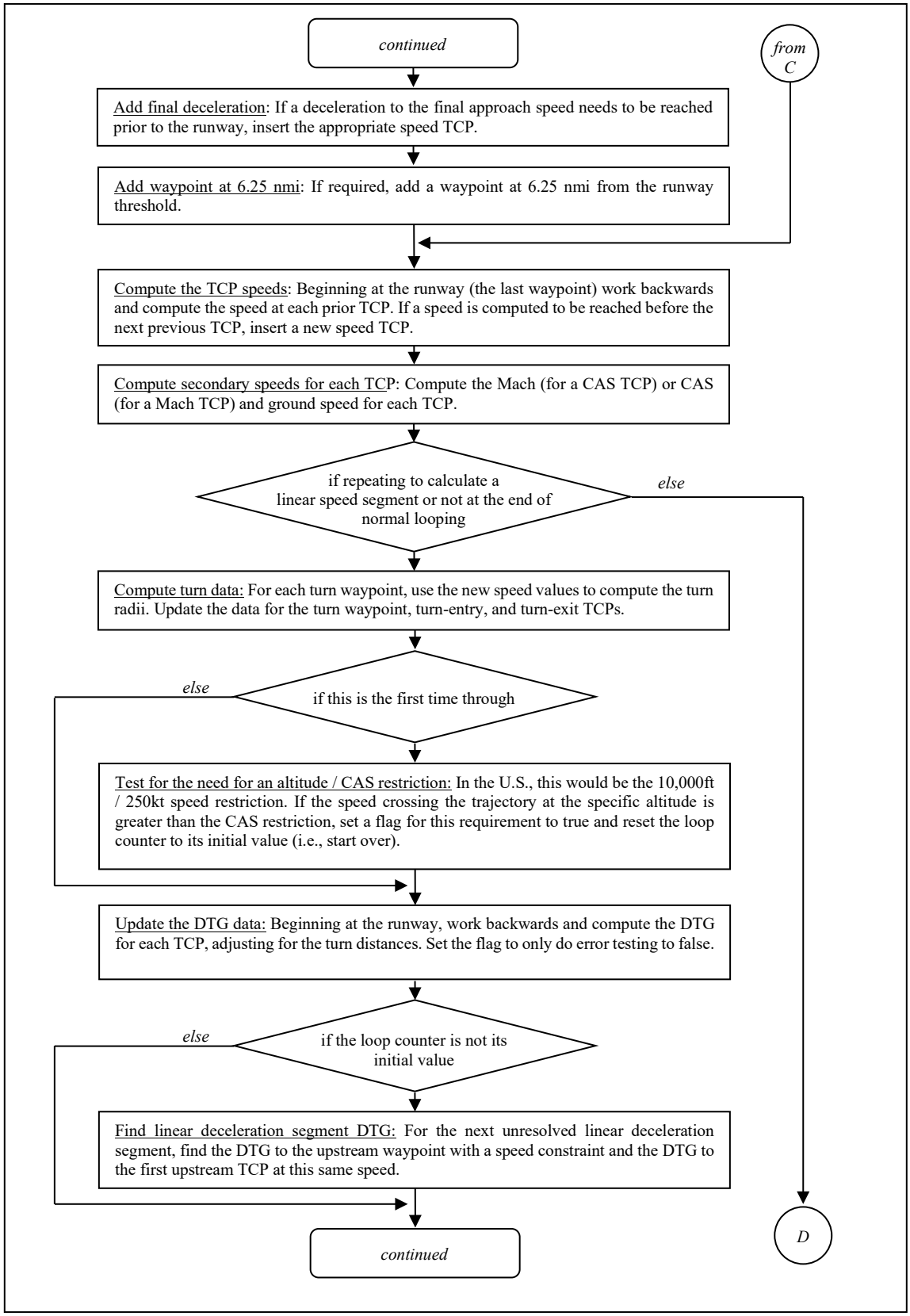


Figure 1 (continued). Basic functions.

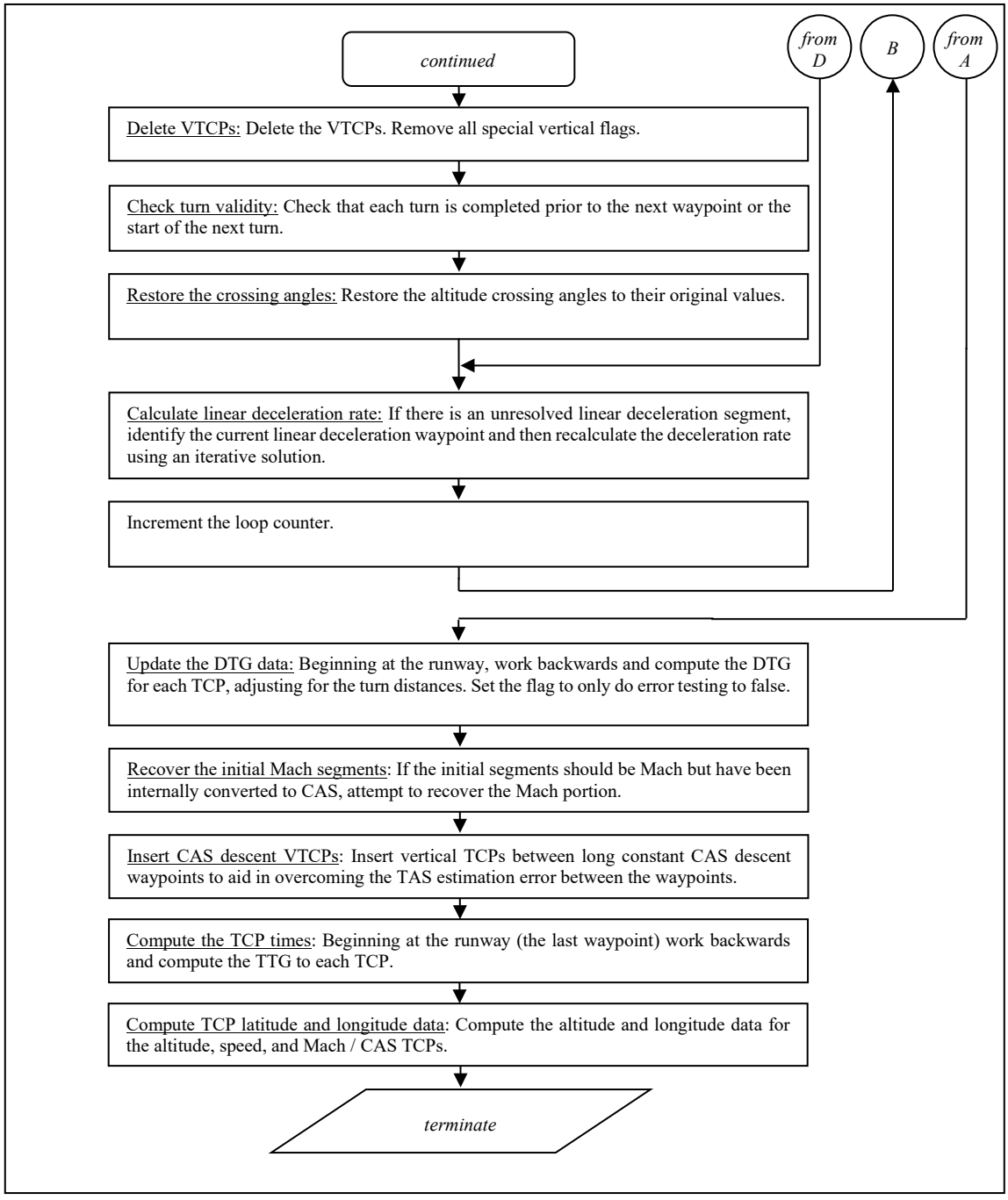


Figure 1 (concluded). Basic functions.

Algorithm Iteration Overview

As noted previously, the major complexity in computing a 4D trajectory involves the interrelationship of ground speed with the path distance around turns. This interrelationship led to an iterative approach in solving for the trajectory speed values. In this implementation, a four-pass iteration was used to calculate the basic, nominal speed values. This latest revision of the algorithm now also accommodates the automatic calculation of the deceleration value to obtain a linear deceleration between two speed-constrained waypoints. To support this latest revision, an extension of the prior iterative technique was developed. For this revision, the following sequence occurs. First, the input data are examined and for all linear deceleration waypoints that are identified, their initial deceleration values are set to the maximum deceleration value allowed by the algorithm, e.g., 5 kt/sec. The trajectory is then calculated using the basic, four-pass iteration loop. The data from these calculations then provide the initial conditions for the calculations for each of the identified linear deceleration waypoint pairs. For each of these waypoint pairs, the calculated distance required to achieve the speed change is compared with the calculated distance between the two waypoints. The deceleration value is then adjusted as necessary to reduce the discrepancy between the two distance calculations. The trajectory is then recalculated until either the discrepancy between the two distance calculations is relatively small or the number of trajectory calculations for that waypoint pair exceeds some maximum number, e.g., 4 iterations. For this latter case, the trajectory calculation would be considered to have failed.

Algorithm Input Data

The algorithm takes as input a list of waypoints, their trajectory-specific data, and associated wind profile data. The list order must begin with the first waypoint on the trajectory and end with the runway threshold waypoint. The trajectory-specific data includes: the waypoint's name and latitude / longitude data, e.g., *Latitude*$_2$ and *Longitude*$_2$, *with the "2" subscript denoting that this is for the second waypoint*; an altitude crossing restriction, if one exists, and its associated crossing angle, e.g., *Crossing Altitude*$_2$ and *Crossing Angle*$_2$; and a speed crossing restriction (Mach or CAS), if one exists, and its associated CAS rate, e.g., *Crossing CAS*$_2$ *and Crossing Rate*$_2$. A value of zero as an input for an altitude or speed crossing constraint denotes that there is no constraint at this point. A special value for the crossing angle, AUTO DESCENT ANGLE, denotes that a linear crossing angle is to be calculated internally by the algorithm. Similarly, if the CAS crossing rate is denoted by the special value of AUTO CAS RATE, then the algorithm is designed to calculate a linear deceleration value. Additionally, a *Crossing Mach* may not occur after any non-zero *Crossing CAS* input. The units for *Crossing Rate* are knots per second.

In this algorithm, a radius-to-fix (RF) segment is indicated by the addition of a center-of-turn position, e.g., *Center of Turn Latitude*$_2$ and *Center of Turn Longitude*$_2$, for the input waypoint at the initiation of the turn. Additional requirements for the RF segment are provided in a subsequent section.

To accommodate a descent from the cruise altitude, a Mach value, *Mach Descent Mach*, may be specified that is different from the cruise Mach value. A CAS value may also be specified for the Mach-to-CAS transition speed, *Mach Transition CAS*, during the descent. Additionally, a CAS speed limit at a defined altitude may also be included. In the U.S., this would typically be set to 250 kt at 10,000 ft.

For routes that terminate at the runway threshold, an input variable, *Final Deceleration Type*, is used to accommodate three different means to achieve the speed at the threshold: RUNWAY, where the final approach speed is met at the runway threshold; STABLE XXXX, where the final approach speed is met at a trajectory altitude value defined in the XXXX variable; and AT FAF, where the final deceleration begins at the final approach fix. To support unusual approach geometries where the final approach fix (FAF) is not the waypoint immediately prior to the runway, the FAF name may be input. Also for routes that terminate at the runway threshold, the input variable *AddMopsRWY625* may be used to invoke the generation of a special waypoint at 6.25 nmi before the landing threshold of the runway. This latter capability to support

this special waypoint at 6.25 nmi before the threshold, along with associated crossing altitude and speed conditions, is a requirement of the RTCA *Minimum Operational Performance Standards (MOPS) for Flight-deck Interval Management (FIM)* (ref. 34).

For the wind forecast, a minimum of two altitude reports (altitude, wind speed, and wind direction) should be provided at each waypoint. The altitudes should span the estimated altitude crossing at the associated waypoint. The algorithm assumes that the input data are valid.

Internal Algorithm Variables

The significant variables computed by this algorithm are as follows:

Data related to the overall path include:

Mach Transition Altitude	*the computed altitude where the transition from Mach to CAS occurs*
NmiToFeet	*6076.115486*

Data specific to the algorithm control logic include:

LDR Mode	*a flag denoting that linear deceleration rate (LDR) calculations are to be performed*
LDR Pending	*a flag denoting that linear deceleration rate (LDR) calculations are to be performed on the next iteration of the algorithm*
Loop Number	*the current algorithm iteration loop number*

Data specific to each TCP include:

Altitude	*the computed altitude at the TCP*
CAS	*the computed CAS at the TCP*
DTG	*the computed, cumulative distance from the last TCP to the TCP*
Ground Speed	*the computed ground speed at the TCP*
Ground Track	*the computed ground track at the TCP*
Mach	*the computed Mach at the TCP*
TTG	*the computed, cumulative time from the last TCP to the TCP*

The algorithm is initiated with only the input waypoints populating the TCP data. There are also several identification variables used by this algorithm for each TCP. These identification variables are noted in the following subsections.

TCPs are denoted in the algorithm in accordance with how they are generated and are marked accordingly in the TCP variable *TcpType*. There are four types of *TcpType* identifiers:

BEGIN TURN	*denotes the beginning point of a turn*
END TURN	*denotes the end point of a turn*
INPUT	*denotes a waypoint from the input data*
VTCP	*denotes a vertical TCP type, VSegType, generated due to a change in the altitude or speed profile*

TCPs are denoted in the algorithm in regard to horizontal path change requirements and are marked accordingly in the TCP variable *TurnType*. There are four types of *TurnType* identifiers:

NO TURN	*the TCP default value*
RF TURN CENTER	*denotes a radius-to fix (RF) center of turn TCP*
TURN END	*denotes the end of a turn*
TURN START	*denotes the start of a turn*

TCPs may also be marked with a vertical identifier, *VSegType*, denoting one of the following:

ALTITUDE	*denotes a change in the descent angle*
ALTITUDE CAS	*denotes a speed change due to a speed restriction at a RESTRICTION specific altitude, e.g., 250 kt at 10,000'*
FINAL SPEED	*the point where the final approach speed deceleration begins*
MACH CAS	*the Mach-to-CAS transition point*
NO TYPE	*the initial VTCP value for a newly created TCP*
RUNWAY625	*the special waypoint at 6.25 nmi before the landing threshold*
SPEED	*denotes a change in the CAS or Mach*
TAS ADJUSTMENT	*an added CAS TCP*
TOD ACCELERATION	*the point where an acceleration to the descent Mach at the top-of-descent occurs*
TOD DECELERATION	*the point where an early transition out of the Mach regime and a deceleration to meet the CAS crossing restriction occurs*
TOD ALTITUDE	*denotes the top-of-descent TCP*

TCPs are also denoted relative to the associated primary speed value, i.e., the crossing speed is Mach or CAS derived.

Additionally, each TCP include data variables required for the calculation of the linear deceleration rate (LDR). These data variables include:

LDR Base Distance	*the distance to the LDR waypoint*
LDR Finished	*a flag denoting that the deceleration estimation has been completed*
LDR Flag	*a flag denoting that the waypoint crossing speed is using a linear deceleration*
LDR Last Rate	*the previously calculated deceleration value*
LDR Last Ratio	*a distance ratio value*
LDR Obtained Distance	*the distance to the previous, upstream, speed-constrained waypoint*
LDR Pass Count	*the number of calculation iterations completed in the estimation*

There are also several input variables that may become overwritten within the algorithm that are required to be restored for subsequent calculation cycles within the algorithm. These variables include the following:

- *Saved Altitude Crossing Angle,* which is the saved input value of *Crossing Angle* for each of the TCP's.
- *Saved Mach Descent Mach*, which is the saved input value of *Mach Descent Mach*.
- *Saved Mach Transition CAS*, which is the saved input value of *Mach Transition CAS*.
- *Saved Mach at First TCP*, which is the saved input Mach value for the first waypoint, i.e., *Crossing* $Mach_{first\ TCP}$, assuming that one exists.

Errors in the Trajectory

The algorithm saves error information related to the ability to generate a valid trajectory. For example, if the deceleration value, *Crossing Rate*, was insufficient to meet the previous speed constraint at waypoint *i*, then an error condition at waypoint *i* would set, i.e., $Error_i$, noting this error situation. There is also an overall error condition, *Fatal Error*, which is set for errors that are typically uncorrectable by a recalculation. In the iteration logic for this algorithm, the error values are reset to a no-error condition at the beginning of each iteration cycle, with the expectation that errors will occur during the refinement of the trajectory values and are only significant at the completion of the iteration cycles. The data logic for error manipulation is not explicitly provided in this documentation, however these error situations will be identified in the text, e.g., "*mark this as an error condition*."

Mach-to-CAS Transitions

General

A significant portion of this algorithm development was devoted to various situations involving the Mach-to-CAS transition. In this regard, the algorithm was designed to accommodate various off nominal, Mach-to-CAS scenarios without the requirement for rigorous, a priori input data development to account for these off-nominal conditions.

Basic Descent Scenario

In the most basic descent scenario for a high performance aircraft, the aircraft would begin the descent at its cruise Mach, descend at that Mach, and then transition to a predetermined CAS at the altitude where the descent Mach and the predetermined CAS represent the same true airspeed value (TAS). The altitude value where this speed equivalence occurs is referred to as the crossover altitude. An example of a basic descent Mach-to-CAS transition is shown in figure 2a, with a cruise and descent Mach values of 0.82 and a transition CAS of 300 kt. The Mach-to-CAS transition altitude in this example occurs at approximately 31,837 ft. Figure 2b portrays the same example with an assumed cruise altitude of 35,000 ft. In figure 2b, the altitude and true airspeed values are plotted with the speed segments labeled appropriately. Figure 2c shows the cruise Mach, descent Mach, and transition CAS segments superimposed over the altitude profile for this scenario.

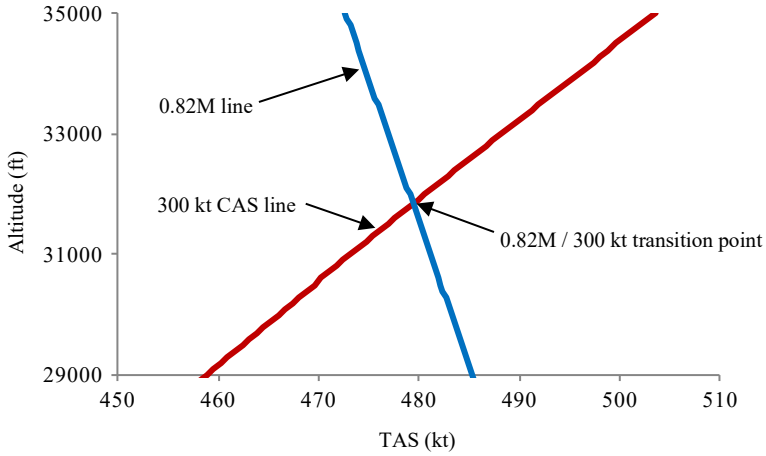


Figure 2a. Example of a basic Mach-to-CAS transition.

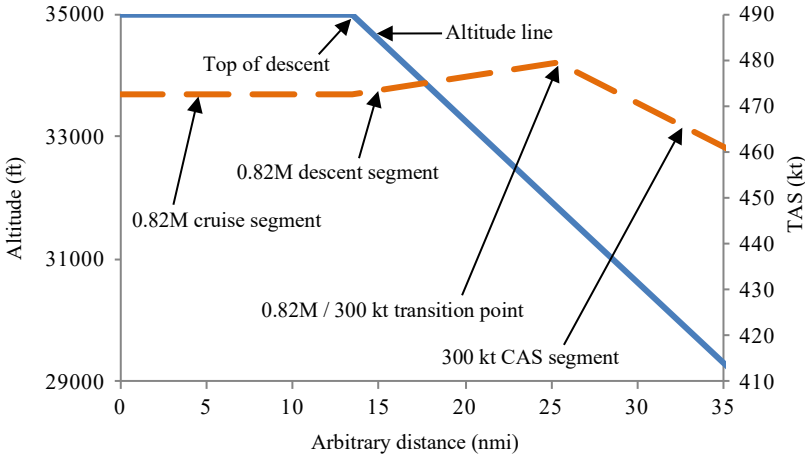


Figure 2b. Mach-to-CAS descent with true airspeed segments.

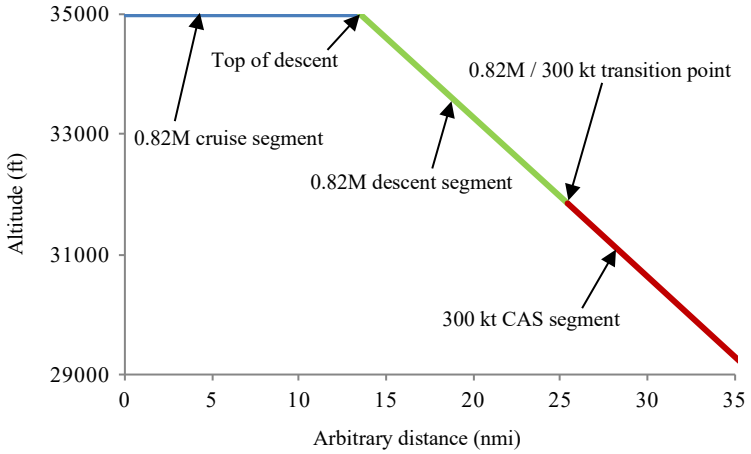


Figure 2c. Speed segments overlaying the altitude profile.

Mach-to-CAS Transition Altitude Above Top of Descent Altitude

The algorithm supports the possibility that the descent Mach and Mach-to-CAS values that were input into the algorithm may result in a Mach-to-CAS transition altitude that is above the cruise altitude. An example of this scenario would be a cruise altitude of 35,000 ft, cruise and descent Mach values of 0.82, and a transition CAS of 270 kt. The Mach-to-CAS transition altitude in this example occurs at approximately 36,503 ft (fig. 3a), 1,503 ft above the top of descent altitude. At the top of descent, the CAS at 0.82M at 35,000 ft is approximately 279 kt. In this scenario, the Mach-to-CAS transition would occur at the top of descent, immediately followed by a deceleration from the 0.82M, 279 kt CAS to the 270 kt CAS descent speed (fig. 3b).

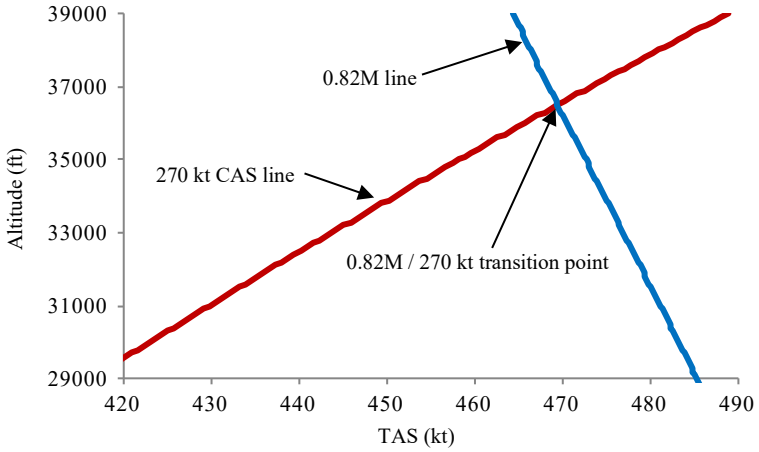


Figure 3a. 0.82 Mach and 270 CAS values.

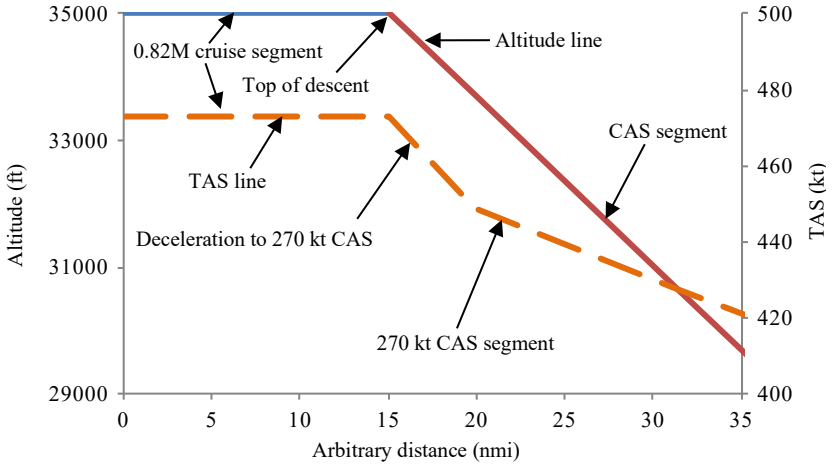


Figure 3b. Mach and CAS segments overlaying the altitude profile.

Descent Mach Greater Than Cruise Mach

The algorithm allows for a descent Mach value that is greater than the cruise Mach, i.e., an acceleration during the initial descent. An example of this scenario would be a cruise altitude of 35,000 ft, a cruise Mach values of 0.80, a descent Mach value of 0.82, and a transition CAS of 300 kt. In this scenario, an acceleration from Mach 0.80 to 0.82 would occur at the top of descent, the acceleration would be completed at approximately 34,588 ft, and the Mach-to-CAS transition altitude would be at approximately 31,837 ft (fig. 4).

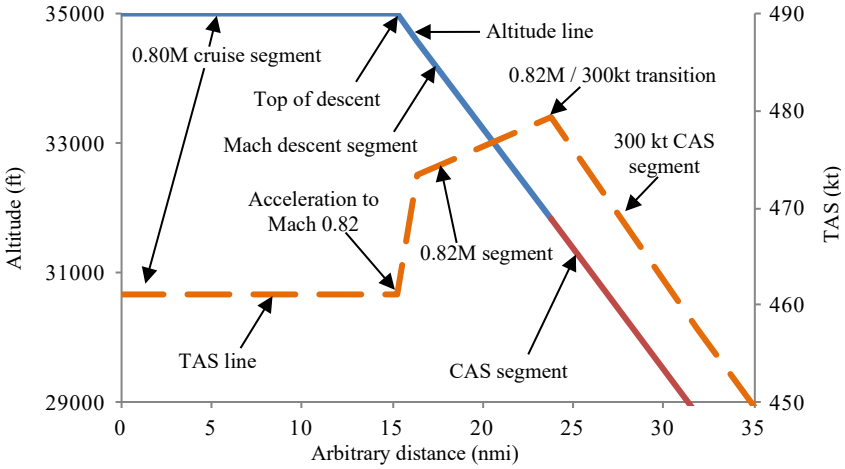


Figure 4. 0.80 Mach cruise, 0.82 Mach descent, and CAS segments overlaying the altitude profile.

Change of Descent CAS to meet a Crossing Restriction

The calculated speed profile may not reach the transition CAS value if attaining that speed would preclude meeting a crossing speed restriction. An example of this scenario is shown in figure 5a. The data for this plot included a cruise altitude of 35,000 ft, a cruise and descent Mach of 0.80M, a planned transition CAS of 300 kt, and a 280 kt CAS constrained waypoint at 29,000 ft. In this scenario, the trajectory would not be able to meet the 300 kt transition speed and then decelerate to the 280 kt crossing speed at the planned deceleration value. In this example, the CAS transition occurs at 31,722 ft and 291 kt and then immediately begins to slow to 280 kt.

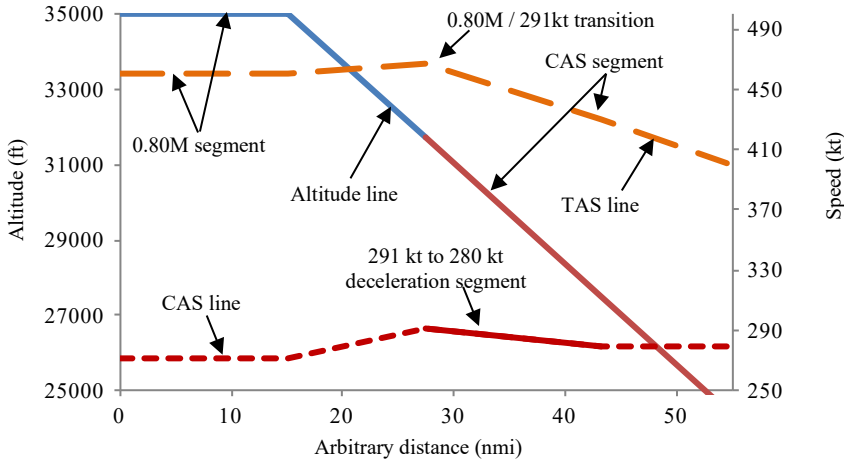


Figure 5a. Descent with CAS speed restriction overlaying the altitude profile.

Another scenario where the calculated speed profile may not reach the transition CAS value due to a crossing speed restriction is shown in figure 5b. The data for this plot included a cruise altitude of 32,000 ft, a cruise and descent Mach of 0.80M, a planned transition CAS of 300 kt, and a 270 kt CAS constrained waypoint at 27,500 ft. In this example, because of the descent angle and the planned deceleration value, the deceleration to meet the crossing restriction occurs at the cruise altitude.

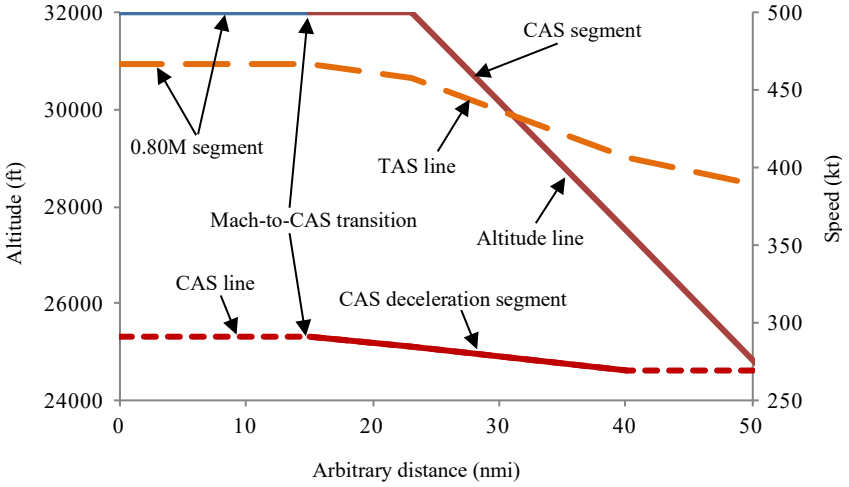


Figure 5b. CAS transition at cruise altitude example.

Description of Major Functions

The functions shown in figure 1 are described in detail in this section. The functions are presented in the order as shown in figure 1. Secondary functions are described in a subsequent section. In these descriptions, the waypoints, which are from the input data and are fixed geographic points, are considered to be TCPs but not all TCPs are waypoints. Nesting levels in the pseudo-code description are denoted by the level of indentation of the document formatting. Additionally, long sections of logic may end with *end of* statements to enhance the legibility of the text.

Preprocess RF Legs

A radius-to-fix (RF) turn segment is a constant radius turn between two waypoints, with lines tangent to the arc around a center of turn point (fig. 6). This function determines if a valid RF turn exists and if so, calculates a pseudo-waypoint relative to the center-of-turn point and inserts it into the waypoint list. The calculated pseudo-waypoint then allows the remainder of the turn calculations performed by this algorithm to be processed as a standard turn. This function is performed in the following manner:

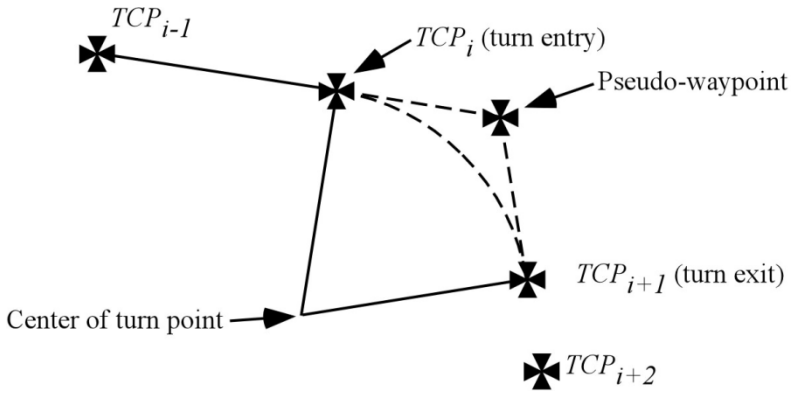


Figure 6. Example of an RF turn.

error = false

Big Turn Error = false

A set of RF turn waypoints is identified by the inclusion of a non-zero value for the latitude and longitude for the center of turn point in the data for the RF turn initiation waypoint. Because three waypoints are needed in an RF turn calculation, two each for the determination of the inbound and outbound track angles, testing is only performed to the next to the last TCP.

for (i = index number of the first TCP + 1; i ≤ index number of the last TCP - 1; i = i + 1)

Determine if this is an RF turn waypoint via the inclusion of the turn center's latitude and longitude data.

if ((Center Of Turn Latitude$_i$ ≠ 0) and (Center Of Turn Longitude$_i$ ≠ 0)) then

Determine the turn direction.

a_1 = *arctangent2(sine(Longitude$_i$ - Longitude$_{i-1}$) * cosine(Latitude$_i$), cosine(Latitude$_{i-1}$) * sine(Latitude$_i$) - sine(Latitude$_{i-1}$) * cosine(Latitude$_i$) * cosine(Longitude$_i$ - Longitude$_{i-1}$))*

a_3 = *arctangent2(sine(Longitude$_{i+1}$ - Longitude$_i$) * cosine(Latitude$_{i+1}$), cosine(Latitude$_i$) * sine(Latitude$_{i+1}$) - sine(Latitude$_i$) * cosine(Latitude$_{i+1}$) * cosine(Longitude$_{i+1}$ - Longitude$_i$))*

deltax = DeltaAngle(a_1, a_3)

where the secondary function *DeltaAngle* is described in a subsequent section.

If *deltax* is positive, this is a right-hand turn.

if (deltax ≥ 0) TurnSign = 1

else TurnSign = -1

Calculate the instantaneous angle at the ending waypoint.

a_2 = *arctangent2(sine(Longitude$_{i+1}$ - Center Of Turn Longitude$_i$) * cosine(Latitude$_{i+1}$),*
*cosine(Center Of Turn Latitude$_i$) * sine(Latitude$_{i+1}$) - sine(Center Of Turn Latitude$_i$) **
*cosine(Latitude$_{i+1}$) * cosine(Longitude$_{i+1}$ - Center Of Turn Longitude$_i$)) +*
*TurnSign * 90°*

Adjust a_2 such that $0° \geq a_2 \geq 360°$.

AdjustAngle(a_2)

deltaa = DeltaAngle(a_1, a_2)

Correct the *deltaa* value if it is in the wrong direction.

if ((TurnSign > 0) and (deltaa < 0°)) then

deltaa = deltaa + 360°

else if ((TurnSign < 0) and (deltaa > 0°)) then

deltaa = deltaa - 360°

If the turn is greater than 170°, break it into two parts so that the standard turn calculations can be performed.

if (|deltaa| > 170°) BigTurn = true

If the turn is less than 3° or more than 260°, it is in error.

if ((|deltaa| < 3°) or (|deltaa| > 260°)) error = true

Perform a center-of-turn test.

if (error = false) then

The radius for point 1 must equal the radius for point 2.

r_1 = *arccosine(sine(Center Of Turn Latitude$_i$) * sine(Latitude$_i$) +*
*cosine(Center Of Latitude$_i$) * cosine(Latitude$_i$) * cosine(Center Of Turn Longitude$_i$ -*
Longitude$_i$))

r_2 = *arccosine(sine(Center Of Turn Latitude$_i$) * sine(Latitude$_{i+1}$) +*
*cosine(Center Of Turn Latitude$_i$) * cosine(Latitude$_{i+1}$) **
cosine(Center Of Turn Longitude$_i$ - Longitude$_{i+1}$))

The radii are considered not equal if the difference is greater than 200 ft. The overall RF leg is considered in error if the turn radius is greater than 10 nmi.

if ((|r_1 - r_2| > (200 / NmiToFeet)) or (r_1 > 10)) error = true

if (error = false) then

If the turn is greater than 170°, generate two waypoints, otherwise, just generate one waypoint.

if (BigTurn) n = 2

else n = 1

*a = TurnSign * 90°*

for (k = 1; k ≤ n; k = k + 1)

Calculate the pseudo-RF waypoint.

The following is the angle from the turn center toward the pseudo waypoint.

$a_3 = a_1 - a$

Adjust a_3 such that $0° \geq a_3 \geq 360°$.

AdjustAngle(a_3)

if (BigTurn) then

if (k = 1) $a_{1b} = a_3 + 0.25 * deltaa$

else $a_{1b} = a_3 + 0.75 * deltaa$

else

There is just one new waypoint, split the turn in half.

$a_{1b} = a_3 + 0.5 * deltaa$

Adjust a_{1b} such that $0° \geq a_{1b} \geq 360°$.

AdjustAngle(a_{1b})

if (k = 1) then

RadialRadialIntercept(Latitude$_i$, Longitude$_i$, a_1,
Center Of Turn Latitude$_i$, Center Of Turn Longitude$_i$, a_{1b},
Latitude$_{rf}$, Longitude$_{rf}$),

noting that *Latitude$_{rf}$ and Longitude$_{rf}$* are returned values.

else

$RadialRadialIntercept(Latitude_{i+1}, Longitude_{i+1}, a_2 + 180°, Center\ Of\ Turn\ Latitude_{i-1}, Center\ Of\ Turn\ Longitude_{i-1}, a_{1b}, Latitude_{rf}, Longitude_{rf})$,

The new waypoint is inserted at location $i+1$ in the waypoint list. This inserted waypoint will appear as an input waypoint to the remainder of the algorithm. The waypoint is inserted between $waypoint_i$ and $waypoint_{i+1}$ from the original list. The function *InsertTcp* should be appropriate for the actual data structure implementation of this function.

InsertTcp(i + 1)

Note that TCP_{i+1} is the newly created waypoint.

Mark TCP_{i+1} as though it was an input waypoint.

$TcpType_{i+1} = INPUT$

Give TCP_{i+1} a unique name.

Also mark this waypoint as a special, RF turn center TCP. This special marking is used in subsequent sections to denote that the center-of-turn point has already been calculated.

$TurnType_{i+1} = RF\ TURN\ CENTER$

$Latitude_{i+1} = Latitude_{rf}$

$Longitude_{i+1} = Longitude_{rf}$

Copy the wind data from TCP_i, the RF initiation waypoint, to TCP_{i+1}, the pseudo-waypoint.

Save the center of turn data. The Turn Data values are associated with each waypoint or TCP record and contain, if appropriate, data relating to turn conditions for that TCP.

$Turn\ Data\ Center\ Latitude_{i+1} = Center\ Of\ Turn\ Latitude_i$

$Turn\ Data\ Center\ Longitude_{i+1} = Center\ Of\ Turn\ Longitude_i$

Increment *i* because a TCP was added and the new TCP at $i + 1$ should not be processed again.

$i = i + 1$

end of for ($k = 1$; $k \leq n$; $k = k + 1$)

end of if (error = false)

end of if ((Center Of Turn Latitude$_i$ ≠ 0) and (Center Of Turn Longitude$_i$ ≠ 0))

end of for (i = index number of the first TCP + 1; ...)

Preprocess Linear Deceleration Data

This is an initialization function that for each waypoint identified as a linear deceleration waypoint, marks it as such and sets the initial deceleration to the maximum deceleration allowable value. The function is performed in the following manner:

for (i = index number of the first TCP; i ≤ index number of the last TCP; i = i + 1)

If the input value for the CAS crossing rate is set to the special value of AUTO CAS RATE, then the algorithm is expected to calculate a linear deceleration value between the two speed constrained waypoints. By design, other trajectory calculations are performed prior to these specific calculations, where these other trajectory calculations require some relatively valid speed calculations. The support these other calculations, the CAS crossing rates for these AUTO CAS RATE segments is initially set to the maximum CAS rate allowed by the algorithm. In this regard, the following calculations are performed:

if (Crossing Cas$_i$ = AUTO CAS RATE) then

LDR Flag$_i$ = true

Set the initial CAS crossing rate to the maximum allowable CAS crossing rate. For this implementation, the maximum allowable CAS crossing rate is 5 kt/sec.

Crossing Rate$_i$ = Maximim Crossing Rate

The following distance variables are used in the iterative calculation for the CAS crossing rate and are initialized to an invalid distance.

LDR Base Distance$_i$ = -1

LDR Obtained Distance$_i$ = -1

else

LDR Flag$_i$ = false

Save Selected Input Data

This is an initialization function that saves the original input values for the altitude crossing angle of each waypoint, the Mach for the first TCP, the descent Mach, and descent CAS. These values are saved because the input values may be overwritten internal to the algorithm and will need to be reset to their original values for each iterative loop. The function is performed in the following manner:

for (i = index number of the first TCP; i ≤ index number of the last TCP; i = i + 1)
Saved Altitude Crossing Angle$_i$ = Crossing Angle$_i$

Saved Mach Descent Mach = Mach Descent Mach

Saved Mach Transition CAS = Mach Transition CAS

Saved Mach at First TCP = Crossing Mach$_{first\ TCP}$

Convert to MSL Altitudes

This is an initialization function that converts altitudes above the *barometric transition altitude* (nominally 18,000'), to an actual, above ground level (above mean sea level) altitude using the waypoint barometric setting from the input data. The function is performed in the following manner:

Initialize the value *Last Altitude* to a very large number.

Last Altitude = -99999

for (i = index number of the last TCP; i ≥ index number of the first TCP; i = i - 1)

Calculate the indicated altitude only if the waypoint has an altitude constraint.

if ((i = index number of the first TCP) or (i = index number of the last TCP) or
(Crossing Angle$_i$ > 0°) or (Crossing Angle$_i$ = AUTO DESCENT ANGLE)) then

if (Crossing Altitude$_i$ > barometric transition altitude) then
Crossing Altitude$_i$ =
ConvertPressureToIndicatedAltitude(Crossing Altitude$_i$, barometric setting$_i$),

where *ConvertPressureToIndicatedAltitude* is a standard aeronautical function to convert pressure altitude to indicated altitude.

if (Crossing Altitude$_i$ < barometric transition altitude)
Crossing Altitude$_i$ = barometric transition altitude

if (Crossing Altitude$_i$ < LastAlt) Crossing Altitude$_i$ = LastAlt

LastAlt = Crossing Altitude$_i$

Generate Initial Tracks and Distances

This is an initialization function that initializes the *Mach Segment* flag, denoting that the speed in this segment is based on Mach, and calculates the point-to-point distances and ground tracks between input waypoints. Great circle equations are used for these calculations, noting that the various dimensional conversions, e.g., degrees to radians, are not shown in the following text.

Generate the initial distances, the center-to-center distances, and ground tracks between input waypoints

for (i = index number of the first TCP; i ≤ index number of the last TCP; i = i + 1)

Start with setting the Mach segments flags to false.

Mach Segment$_i$ = false

Compute the waypoint-center to waypoint-center distances.

if (i = index number of the first TCP) Center to Center Distance$_i$ = 0

else

Center to Center Distance$_i$ =
*arccosine(sine(Latitude$_{i-1}$) * sine(Latitude$_i$) + cosine(Latitude$_{i-1}$) * cosine(Latitude$_i$) * cosine(Longitude$_{i-1}$ - Longitude$_i$))*

Ground Track$_{i-1}$ =
*arctangent2(sine(Longitude$_i$ - Longitude$_{i-1}$) * cosine(Latitude$_i$), cosine(Latitude$_{i-1}$) * sine(Latitude$_i$) - sine(Latitude$_{i-1}$) * cosine(Latitude$_i$) * cosine(Longitude$_i$ - Longitude$_{i-1}$))*

end of for (i = index number of the first TCP; i ≤ index number of the last TCP; i = i + 1)

Now set the runway's ground track.

Ground Track$_{last\ TCP}$ = Ground Track$_{last\ TCP\ -\ 1}$

The cumulative distance, DTG, is computed as follows:

DTG$_{last\ TCP}$ = 0

for (i = index number of the last TCP; i > index number of the first TCP; i = i - 1)

DTG$_{i-1}$ = DTG$_i$ + Center to Center Distance$_i$

Initialize Waypoint Turn Data

The *Initialize Waypoint Turn Data* function is used to determine if a turn exists at a waypoint and if so, inserts turn-entry and turn-exit TCPs. Waypoints that have more than a 3 degree change in ground track between the previous waypoint and the next waypoint are considered turn-waypoints. The function is performed in the following manner:

i = index number of the first TCP + 1

Last Track = Ground Track$_{first\ TCP}$

Note that the first and last TCPs cannot be turns.

while (i < index number of the last TCP)

Track Angle After = Ground Track$_i$

a = DeltaAngle(Last Track, Track Angle After)

Check for a turn that is greater than 170 degrees.

if (|a| > 170°) then

Set an error and ignore the turn.

Mark this as a fatal error condition.

$a = 0°$

If the turn is more than 3-degrees, compute the turn data.

if (|a| > 3°) then

half turn = a / 2

Track Angle Center = Last Track + half turn

This is the center of the turn, e.g., the original input waypoint.

$Ground\ Track_i = Track\ Angle\ Center$

$Turn\ Data\ Track1_i = Last\ Track$

$Turn\ Data\ Track2_i = Track\ Angle\ After$

If this is not an RF turn, then the turn radius needs to be calculated.

$if\ (TurnType_i \neq RF\ TURN\ CENTER)\ Turn\ Data\ Turn\ Radius_i = 0$

$Turn\ Data\ Path\ Distance_i = 0$

Insert a new TCP at the end of the turn.

The new TCP is inserted at location *i+1* in the TCP list. The TCP is inserted between TCP_i and TCP_{i+1} from the original list. The function *InsertTcp* should be appropriate for the actual data structure implementation of this function.

InsertTcp(i + 1)

Note that TCP_{i+1} is the new TCP.

$TcpType_{i+1} = END\ TURN$

$DTG_{i+1} = DTG_i$

$Ground\ Track_{i+1} = Track\ Angle\ After$

The start of the turn TCP is as follows,

InsertTcp(i)

$TcpType_i = BEGIN\ TURN$

Note that the original TCP is now at index i + 1.

$DTG_i = DTG_{i+1}$

$Ground\ Track_i = Last\ Track$

$Last\ Track = Track\ Angle\ After$

$i = i + 2$

end of if (|a| > 3°)

else $Last\ Track = Ground\ Track_i$

$i = i + 1$

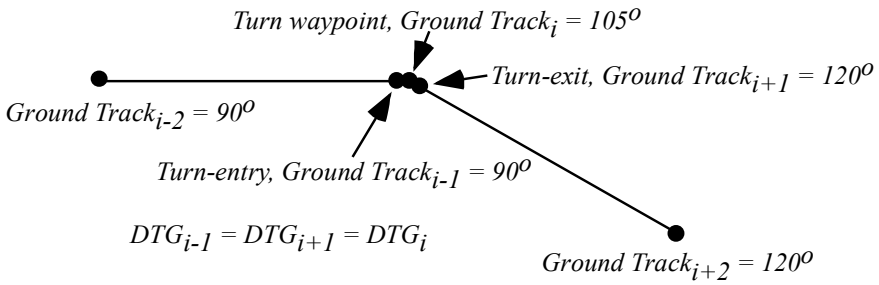
end of while (i < index number of the last TCP)

Effectively, this function:

- Marks each turn-waypoint and sets its ground track angle to the computed angle at the midpoint of the turn.
- Inserts a co-distance turn-entry TCP before this turn-waypoint with the ground track angle for this turn-entry TCP set to the value of the inbound ground track angle.
- Inserts a co-distance turn-exit TCP after this turn-waypoint with the ground track angle for this turn-exit TCP set to the value of the outbound ground track angle.

An example illustrating the inserted turn-start and turn-end TCPs is shown in figure 7.

Figure 7. Initialized turn waypoint.



Determine Linear Deceleration Requirements

This routine is used in the linear deceleration rate calculations to determine if additional passes are needed to accommodate linear deceleration rate calculations. External variables used by this routine include the current algorithm iteration loop number, *Loop Number*, the LDR mode flag, *LDR Mode*, and the flag denoting that LDR calculations are pending, *LDRPending*.

On the first call to this routine, determine if any LDR waypoints exist, and, if so, set the *LDR Pending* flag to true.

if (Loop Number == basic, four-pass iteration loop value)

Determine if any LDR waypoints exist, and if so, set the *LDR Pending* flag to true. This determination uses the secondary function *FindNextLDRWaypoint*, described in a subsequent section.

if (FindNextLDRWaypoint() > 0) LDR Pending = true

else

Find the next linear deceleration rate waypoint whose speed has not been calculated using the secondary function *FindNextLDRWaypoint*.

idx = FindNextLDRWaypoint()

Check for errors using the function *LDRRateCheck*, which will return a value of true for the variable *LdrRateError* if the current linear deceleration segment is not valid and will return a value of true for the variable *OtherError* if a valid speed could not be calculated for any segment. Note that the variable *LdrRateError* is not used in *Determine Linear Deceleration Requirements* but is used in other functions.

OtherError = true

if (idx > -1) LDRRateCheck(idx, LDR Pending, LdrRateError, OtherError)

if ((idx > -1) and (OtherError = false)) then

if (LDR Pending)

Change the values of the LDR Pending and LDR Mode.

LDR Pending = false

LDR Mode = true

else

The LDR Mode is no longer valid.

LDR Pending = false

LDR Mode = false

Reset the Descent Speed Values

The *Reset the Descent Speed Values* function simply replaces the current values for *Mach Descent Mach, Mach Transition CAS*, and *Crossing Mach*$_{first\ TCP}$ with the values that were saved in the function *Save Selected Input Data* and reset the descent speed flags *MachCasAtTod* and *AllowTodDeceleration* to false.

Compute TCP Altitudes

Beginning with the last waypoint, the *Compute TCP Altitudes* function computes the altitudes at each previous TCP and inserts any additional altitude TCPs that may be required to denote a change in the altitude profile. The function uses the current altitude constraint (TCP_i in fig. 8), searches backward for the previous constraint (TCP_{i-3} in fig. 8), and then computes the distance required to meet this previous constraint. The altitudes for all of the TCPs within this distance are computed and added to the data for the TCPs. If the along-path distance to meet the previous constraint is not at a TCP, a new altitude VTCP is inserted at this distance. An example of this is shown in figure 9. In addition, if the *Crossing Angle* for a waypoint is set to -99, this denotes that the algorithm is to internally compute the *Crossing Angle* between this and the next higher, altitude constrained waypoint, noting that this option should only be used in situations where the relevant waypoint pairs are known to procedurally have a fixed angle between them. This function is performed in the following steps:

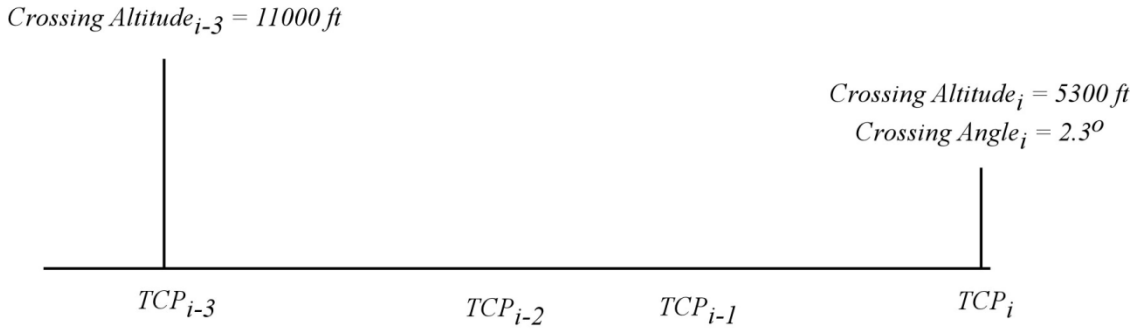


Figure 8. Input altitude crossing constraints.

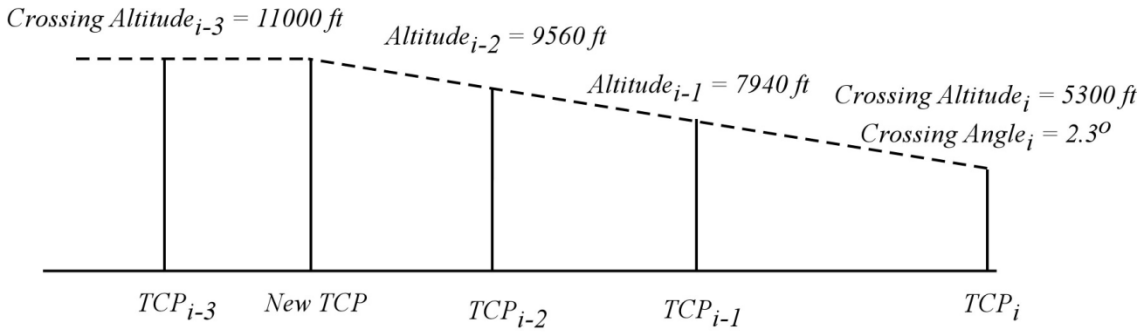


Figure 9. Computed altitude profile with TCP added.

Set the current constraint index number, *cc*, equal to the index number of the last TCP,

cc = index number of the last TCP

Set the altitude of this waypoint to its crossing altitude,

$Altitude_{cc} = Crossing\ Altitude_{cc}$

Set a flag denoting that the TOD point has not been identified.

Have TOD = false

While (cc > index number of the first TCP)

If this is the TOD, mark this point.

if ((Have TOD = false) and ($Altitude_{cc} \geq Altitude_{1}$)) then

Mark this as the TOD point.

Have TOD = true

$VSegType_{cc}$ = *TOD ALTITUDE*

Determine if the previous constraint cannot be met.

If ($Altitude_{cc} > Crossing\ Altitude_{cc}$) *then*

The constraint has not been made.

If this is the last pass through the algorithm, mark this as a fatal error condition.

$Altitude_{cc} = Crossing\ Altitude_{cc}$

Find the prior waypoint index number *pc* that has an altitude constraint, e.g., a crossing altitude ($Crossing\ Altitude_{pc} \neq 0$). This may not always be the previous (i.e., *cc - 1*) waypoint.

Initial condition is the previous TCP.

pc = cc - 1

while ((pc > index number of the first TCP) and ($TcpType_{pc} \neq INPUT$) *or*
($Crossing\ Altitude_{pc} = 0$))) *pc = pc - 1*

Save the previous crossing altitude,

$Prior\ Altitude = Crossing\ Altitude_{pc}$

Save the current crossing altitude (*Test Altitude*) at TCP_{cc} and the descent angle (*Test Angle*) noting that the first and last waypoints always have altitude constraints and except for the first TCP, all constrained altitude points must have descent angles.

$Test\ Altitude = Crossing\ Altitude_{cc}$

$Test\ Angle = Crossing\ Angle_{cc}$

If the Test Angle value, i.e., AUTO DESCENT ANGLE, denotes that this is angle is to be computed internally as a linear descent between the two altitude constrained waypoints then the following calculations are performed:

if (Test Angle = AUTO DESCENT ANGLE) then

$dx = DTG_{pc} - DTG_{cc}$

dy = Prior Altitude - Test Altitude

*Test Angle = arctangent2 (dy, NmiToFeet * dx)*

Crossing Angle$_{cc}$ = Test Angle

Test for an extreme angle, e.g., 7.5°.

if (Test Angle > maximum allowable descent angle) mark this as a fatal error condition.

Compute all of the TCP altitudes between the current TCP and the previous crossing waypoint.

k = cc

while (k > pc)

If the previous altitude has already been reached, set the remaining TCP altitudes to the previous altitude.

if (Prior Altitude ≤ Test Altitude) then

for (k = k - 1; k > pc; k = k - 1) Altitude$_k$ = Test Altitude

Set the altitude at the last test point.

Altitude$_{pc}$ = Test Altitude

else

Compute the distance from *TCP$_k$* to the *Prior Altitude* using the altitude difference between the *Test Altitude* and the *Prior Altitude* with the *Test Angle*. If there is no point at this distance, add a TCP at that distance.

Compute the distance *dx* to make the altitude.

if (Test Angle ≤) dx = 0

*else dx = (Prior Altitude - Test Altitude) / (NmiToFeet * tangent(Test Angle))*

Compute the altitude *z* at the previous TCP.

*z = ((DTG$_{k-1}$ - DTG$_k$) * NmiToFeet) * tangent(Test Angle) + Test Altitude*

If there is a TCP prior to this distance or if *z* is very close to the *Prior Altitude*, compute and insert its altitude.

if ((DTG$_{k-1}$ < (DTG$_k$ + dx)) or (|z - Prior Altitude| < some small value)) then

if (|z - Prior Altitude| < some small value) Altitude $_{k-1}$ = Prior Altitude

else Altitude $_{k-1}$ = z

Check to see if the constraint has been reached with a 100 ft tolerance; if not, set an error condition.

if ((k-1) = pc) then

if (|Altitude$_{pc}$ - Crossing Altitude$_{pc}$| > 100ft) mark this as a fatal error condition

Always set the crossing exactly to the crossing value.

Altitude$_{pc}$ = Crossing Altitude$_{pc}$

Update the Test Altitude.

Test Altitude = Altitude $_{k-1}$

Decrement the counter to set it to the prior TCP.

k = k - 1

end of if ((DTG$_{k-1}$ < (DTG$_k$ + dx)) or (|z - Prior Altitude| < some small value))

else

The altitude constraint is reached prior to the TCP, a new VTCP will need to be inserted at that point. The distance to the new TCP is,

d = DTG$_k$ + dx

Compute the ground track at distance *d* along the trajectory and save it as *Saved Ground Track*.

Saved Ground Track = GetTrajGndTrk(d)

Insert a new VTCP at location *k* in the TCP list. The VTCP is inserted between TCP$_{k-1}$ and TCP$_k$ from the original list. The function *InsertTcp* should be appropriate for the actual data structure implementation of this function.

InsertTcp(k)

Update the data for the new VTCP which is now *TCP$_k$*.

TcpType$_k$ = VTCP

if (VSegType$_k$ = NO TYPE) VSegType$_k$ = ALTITUDE

DTG$_k$ = d

Altitude$_k$ = Prior Altitude

if ((Have TOD = false) and (Altitude$_k$ ≥ Altitude$_{first\ TCP}$)) then

Have TOD = true

VSegType$_k$ = TOD ALTITUDE

Add the ground track data which must be computed if the new VTCP occurs within a turn. The functions *WptInTurn* and *ComputeGndTrk* are described in subsequent sections.

if (WptInTurn(k)) Ground Track$_k$ = ComputeGndTrk(k, d)

else Ground Track$_k$ = Saved Ground Track

Compute and add the wind data at distance *d* along the path to the data of *TCP$_k$*.

GenerateWptWindProfile(d, TCP$_k$)

Test Altitude = Prior Altitude

Since *TCP$_k$*, has now been added prior to *pc*, the current constraint counter *cc* needs to be incremented by 1 to maintain its correct position in the list.

cc = cc + 1

end of else

end of else *if (Prior Altitude ≤ Test Altitude)*

The function loops back to *while (k > pc).*

end of while (k > pc)

Now go to the next altitude change segment on the profile.

cc = k

The function loops back to *while cc > index number of the first TCP.*

end of while (cc > index number of the first TCP).

Copy Crossing Angles

The *Copy Crossing Angles* is a simple function that starts with the next to last TCP and copies the subsequent crossing angle if the current TCP does not have a crossing angle. E.g.,

for (i = index number of the last TCP - 1; i ≥ index number of the first TCP; i = i - 1)

if (Crossing Angle$_i$ = 0) Crossing Angle$_i$ = Crossing Angle$_{i+1}$

Evaluate the Descent Mach/CAS

The *Evaluate the Descent Mach/CAS* function evaluates the validity of the Mach/CAS transition speed requirement. If the Mach/CAS transition altitude, based on the Mach/CAS transition speeds, is above the

cruise altitude, this function will remove the Mach/CAS transition speed requirement since a transition to CAS is required at the TOD.

Only perform this evaluation if the path begins with a Mach defined waypoint and a Mach-to-CAS transition speed exists.

if ((Crossing Mach $_{first\ TCP} \neq 0$) and (Mach Transition CAS $\neq 0$)) then

Find the top of descent.

FoundTod = false

i1 = index number of the first TCP

m = Crossing Mach $_{first\ TCP}$

while ((FoundTod = false) and (i1 < index number of the last TCP))

if (VSegType$_{i1}$ = TOD ALTITUDE) FoundTod = true

else i1 = i1 + 1

Find the last Mach before the TOD.

FoundLastMach = false

i2 = 0

while ((FoundLastMach = false) and (i2 ≤ i1))

if (Crossing Mach$_{i2}$ > 0) then

FoundLastMach = true

m = Crossing Mach$_{i2}$

else i2 = i2 + 1

Determine if there is a Mach crossing waypoint after the TOD.

FoundLaterMach = false

i3 = i2 + 1

while ((FoundLaterMach = false) and (i3 < index number of the last TCP))

if (Crossing Mach$_{i3}$ > 0) FoundLaterMach = true

else i3 = i3 + 1

If the speeds are valid, calculate the Mach/CAS transition altitude and compare it against the TOD altitude.

if (FoundTod and FoundLastMach and (FoundLaterMach = false)) then

Get the slowest, valid Mach value from the input data.

if ((Mach Descent Mach > 0) and (Mach Descent Mach < m)) m = Mach Descent Mach

Invoke the secondary function *FindMachCasTransitionAltitude* which calculates the altitude where the Mach and CAS are equal.

z = FindMachCasTransitionAltitude(Mach Transition CAS, m)

if (z >= (Altitude$_{i1}$ - 1ft)) then

AllowTodDeceleration = true

Mach Descent Mach = 0

end of if (FoundTod and FoundLastMach and (FoundLaterMach = false))

end of if ((Crossing Mach $_{first\ TCP}$ ≠ 0) and (Mach Transition CAS ≠ 0))

Meet Cruise CAS Restriction

The *Meet Cruise CAS Restriction* function changes, if required, the descent Mach if there is a high altitude, CAS restricted waypoint and the computed speed is above the required crossing speed for that CAS waypoint.

The calling function provides as input and retains the subsequent outputs for the following variables: *TodId, TodMach, TodMachRate, MachCasAtTod, and AllowTodDeceleration*. The variable *TodId* is the name of the top-of-descent waypoint (TOD) and is initialized as an empty string by the calling program. This *Meet Cruise CAS Restriction* function may modify the Mach and speed change rate that occurs at the TOD, *TodMach* and *TodMachRate*, respectively, and these values are then passed to subsequent functions that require these data. The variable *MachCasAtTod* is a flag that if true, indicates that the Mach-to-CAS transition occurs at the TOD point. This variable is used by the functions *Change TOD Mach Value* and *Compute Mach-to-CAS TCP*.

If the input Mach value for the first TCP is not valid, i.e., the path does not start with a Mach segment, the function terminates with *MachCasAtTod* set to false. Otherwise, the following is performed.

if (Crossing Mach $_{first\ TCP}$ = 0) terminate this function. Otherwise,

Set the initial values.

MachCasAtTod = false

MachCasModified = false

CasIndex = index number of the first TCP

TodIndex = index number of the first TCP

AltAtMach = 0.

LastMach = 0

z = 0

done = false

If the TOD Mach data have been modified in a previous invocation of *Change TOD Mach Value*, indicated by a non-empty value for *TodId*, reset their values.

if (TodId ≠ empty) then

fini = false

i = index number of the first TCP

Find the waypoint with the name defined in *TodId.*

while ((i ≤ (index number of the last TCP)) and (fini = false))

if (Id_i = TodId) then

fini = true

Crossing $Mach_i$ = TodMach

Crossing CAS_i = 0

Crossing $Rate_i$ = TodMachRate

TodId = empty string

i = i + 1

end of if (TodId ≠ empty)

Find the first CAS waypoint.

fini = false

i = index number of the first TCP

while ((i ≤ index number of the last TCP) and (fini = false))

if (Crossing CAS_i > 0) then

CasIndex = i

fini = true

i = i + 1

Determine if the trajectory is already at the CAS altitude, i.e., the initial altitude is the CAS altitude, and if so, start in a CAS mode, not Mach.

if (Crossing $Altitude_{first\ TCP}$ *=* $Altitude_{CasIndex}$*) then*

done = true

for (k = index number of the first TCP; k < CasIndex; k = k + 1)

if (Crossing $Mach_k > 0$*) then*

Change the route data so that the trajectory is starting in a CAS mode.

Invoke the secondary function *MachToCas*. This function is described in a subsequent section.

$Crossing\ CAS_k = MachToCas(Crossing\ Mach_k, Altitude_{CasIndex})$

$Crossing\ Mach_k = 0$

$Mach\ Segment_k = false$

end of if (Crossing $Mach_k > 0$*)*

if (done = false) then

Find the last Mach value.

fini = false

i = index number of the first TCP

while ((i < index number of the last TCP) and (fini = false))

if (Crossing $CAS_i > 0$*) fini = true*

else if (Crossing $Mach_i > 0$*) LastMach = Crossing* $Mach_i$

i = i + 1

Determine the descent Mach value.

if (Mach Descent Mach ≠ 0) DescentMach = Mach Descent Mach

else DescentMach = LastMach

Determine the Mach-to-CAS transition CAS value.

if (Mach Transition CAS > 0) then

MachCas = Mach Transition CAS

if (Mach Transition CAS < Crossing $CAS_{CasIndex}$) MachCas = Crossing $CAS_{CasIndex}$

else MachCas = Crossing $CAS_{CasIndex}$

Find the last Mach altitude.

fini = false

i = index number of the first TCP

while ((i ≤ index number of the last TCP) and (fini = false))

if (Crossing CAS_i > 0) fini = true

else if (Crossing $Altitude_i$ > 0) AltAtMach = Crossing $Altitude_i$

i = i + 1

Determine if the Mach is slower than the descent CAS. This is a special case.

DoCasDecel = false

No Mach segments or a different descent Mach?

if (AllowTodDeceleration) then

Initially assume that the TOD altitude is the initial altitude.

TodCas = MachToCas(LastMach, $Altitude_{TodIdx}$)

if (TodCas > MachTransitionCas) DoCasDecel = true

else WptRecords->AllowTodDeceleration = false

MachCasAtTod = true

Invoke the secondary function *FindMachCasTransitionAltitude* which calculates the altitude where the Mach and CAS are equal. This function is described in a subsequent section.

z = FindMachCasTransitionAltitude(MachCas, DescentMach)

if ((z > Crossing $Altitude_{first\ TCP}$) and (DoCasDecel = false)) then

The path is already below the transition altitude, change the route data so it starts in a CAS mode.

for (k = index number of the first TCP; k < index number of the last TCP; k = k + 1)

done = true

if (Crossing Mach$_k$ > 0) then

Crossing CAS$_k$ = MachCas

Crossing Mach$_k$ = 0

Mach Segment$_k$ = false

end of if ((z > Crossing Altitude$_{first\ TCP}$) and (DoCasDecel = false))

end of if (done = false)

if (done = false) then

If the path still starts with a Mach segment, which may have already been modified in this function, test for other special cases.

If required, handle the special case of an accelerated descent.

if (DescentMach > LastMach) then

Invoke the secondary function *ComputeDescentAccelDecel*. This function handles the special case of a Mach acceleration in the descent where the first CAS crossing restriction cannot be met. This function is described in a subsequent section. This function may modify the waypoint data.

ComputeDescentAccelDecel(CasIndex, LastMach, MachCasModified, DescentMach, MachCas)

If the descent data are changed, recalculate *z*.

if (MachCasModified) then

z = FindMachCasTransitionAltitude (MachCas, DescentMach)

Next, update the waypoint data.

Mach Descent Mach = DescentMach

Mach Transition CAS = MachCas

end of if (DescentMach > LastMach)

if (z < Crossing Altitude$_{CasIndex}$) then

At this point, the descent CAS or Mach needs to be changed.

If the descent CAS is faster than the crossing CAS, determine if changing the descent CAS corrects the problem.

fini = false

if (MachCas > Crossing $CAS_{CasIndex}$*) then*

s = MachToCas(DescentMach, $Altitude_{CasIndex}$*)*

if (s >= Crossing $CAS_{CasIndex}$*) then*

MachCas = s

Mach Transition CAS = s

fini = true

m = CasToMach(MachCas, $Altitude_{CasIndex}$*)*

if ((fini = false) and (m > DescentMach)) then

s = MachToCas(DescentMach, $Altitude_{CasIndex}$*)*

if (s >= Crossing $CAS_{CasIndex}$*) then*

Change to descent CAS.

MachCas = s

Mach Transition CAS = s

else

Change the descent Mach.

if (MachCas ≤ Crossing $CAS_{CasIndex}$*)*

DescentMach = CasToMach(MachCas, $Altitude_{CasIndex}$*)*

else DescentMach = CasToMach(Crossing $CAS_{CasIndex}$*,* $Altitude_{CasIndex}$*)*

else if (fini = false)

DescentMach = CasToMach(MachCas, $Altitude_{CasIndex}$*)*

Mach Descent Mach = DescentMach

z = $Altitude_{CasIndex}$

Perform an extreme limits test, assuming that a valid Mach value will be between 0.6 and 0.9 Mach.

if ((DescentMach > 0.9) or (DescentMach < 0.6)) mark this as a fatal error condition

end of if ((fini = false) and (m > DescentMach))

Make sure that there is sufficient distance to slow from the Mach-to-CAS transition speed to make the crossing CAS.

if ((z ≥ $Altitude_{CasIndex}$) and (MachCas > Crossing $CAS_{CasIndex}$) and (Crossing $Rate_{CasIndex}$ > 0) and (MachCasModified = false)) then

Find the distance at *z*. This is an iterative solution.

i = CasIndex - 1

fini = false

j = index number of the first TCP

Calculate the headwind at the end point. This calculation uses the secondary function *InterpolateWindWptAltitude*, described in a subsequent section.

InterpolateWindWptAltitude(Wind $Profile_{CasIndex}$, $Altitude_{CasIndex}$, Ws, Wd, Td)

*HeadWind = Ws * cosine(Wd - GndTrack $_{CasIndex}$)*

CurrentGs = ComputeGndSpeedUsingTrack(Crossing $CAS_{CasIndex}$, $GndTrack_{CasIndex}$, $Altitude_{CasIndex}$, Ws, Wd, Td)

Iterate = false

OnePass = true

MachCasHold = MachCas

LastCut = 0

while (fini = false)

i = CasIndex - 1

while ((i > index number of the first TCP) and (Altitude $_i$ < z)) i = i - 1

if (($Altitude_i$ - $Altitude_{i+1}$) ≤ 0) a = 0

else a = (z - $Altitude_{i+1}$) / ($Altitude_i$ - $Altitude_{i+1}$)

Calculate the distance, *dx*, required to reach the altitude.

*dx = a * (DTG_i - DTG_{i+1}) + DTG_{i+1} - $DTG_{CasIndex}$*

InterpolateWindWptAltitude(Wind $Profile_{CasIndex}$, z, Ws2, Wd2, Td2)

*Hw2 = Ws2 * cosine(Wd2 - $GndTrack_i$)*

AvgHw = (HeadWind + Hw2) / 2

Invoke the secondary function *EstimateNextCas*. *EstimateNextCas* is an iterative function to estimate the CAS value at the next waypoint.

CasTest =EstimateNextCas(Crossing $CAS_{CasIndex}$*, CurrentGs, true, MachCasHold, AvgHw, z, dx, Crossing* $Rate_{CasIndex}$*, Td)*

If it is required, set up the iteration values, where these values are in CAS.

if (OnePass = true) then

if (CasTest < MachCas) Iterate = true

else fini = true

OnePass = false

Calculate the iteration step size.

LastCut = |MachCas - CasTest|

Limit the step size to no smaller than 2 kt.

if (LastCut < 2) LastCut = 2

if (Iterate) then

if (MachCas ≥ CasTest) s = MachCas - LastCut

else s = MachCas + LastCut

*LastCut = 0.5 * LastCut*

if (s > MachCasHold) s = MachCasHold

Determine if the Mach-to-CAS estimate is valid.

if (((s + 0.05) ≥ MachCas) and (|s - MachCas| < 0.1)) then

fini = true

Calculate the Mach-to-CAS altitude for the current estimate.

z = FindMachCasTransitionAltitude (MachCas, DescentMach)

Determine if a deceleration is needed prior to the TOD. Add a 50 ft buffer value.

if (z > (AltAtMach + 50 ft)) then

Find the TOD waypoint.

fini2 = false

j = index number of the first TCP

while ((j < index number of the last TCP) and (fini2 = false))

if (Waypoint$_j$ is marked as the TOD point) fini2 = true

else j = j + 1

The altitude index for the test is the TOD altitude point.

if (fini2 and (i = j)) then

Mach Descent Mach = CasToMach(Mach Transition CAS, AltAtMach)

MachCasAtTod = true

end of if (z > (AltAtMach + 50))

end of if (((s + 0.05) ≥ MachCas) and (|s - MachCas| < 0.1))

else

Mach Transition CAS = s

MachCas = s

z = FindMachCasTransitionAltitude(MachCas, DescentMach)

if (z > Altitude$_i$) z = Altitude$_i$

j = j + 1

Add a test to limit the number of iterations to 10.

if (j ≥ 10) fini = true

end of if (Iterate)

end of while (fini = false)

end of if (done = false)

Add TOD Deceleration TCP

This function handles the special case where meeting a CAS restriction at a downstream waypoint requires an early transition out of the Mach regime and a deceleration at or near the top of descent to meet the CAS crossing restriction. This function computes the distance required to meet the CAS crossing restriction and the related speed and altitude values at the distance. A TCP at that distance is then inserted into the trajectory to identify the start of the deceleration segment. This function is only performed if the

input data starts with a Mach *Crossing Speed* for the first TCP and the prior determination that a cruise CAS waypoint requires a TOD deceleration.

The following variables are initialized:

(MachAtTOD)LastMach = Crossing Mach$_{first\ TCP}$

fini = false

(TodIndex)TestIndex = index number of the first TCP

i = index number of the first TCP

Find the TOD waypoint.

while ((i < index number of the last TCP) and (fini = false))

if (Crossing Mach$_i$ > 0) LastMach = Crossing Mach$_i$

if ((Altitude$_i$ < Altitude$_{first\ TCP}$) or (Crossing CAS$_i$ > 0))

if (Altitude$_i$ ≠ Altitude$_{first\ TCP}$) TestIndex = i - 1

else TestIndex = i

fini = true

i = i + 1

end of while ((i < index number of the last TCP) and (fini = false))

Make an initial estimate of the distance to the deceleration CAS value. The function *TodDecelerationDistance* returns the values *Valid, k*, and *dx*.

TodDecelerationDistance(TestIdx, LastMach, Mach Transition CAS, Valid, k, dx)

Since the normal descent Mach-to-CAS transition will not occur, the start of deceleration TCP is added here.

InsertTcp(TestIndex +1)

Update the data for the new TCP which is now *TCP$_{TestIndex+1}$*.

Copy all of the data from TCP$_{TestIndex}$ into TCP$_{TestIndex\ +1}$

Now set the data in *TCP$_{TestIndex\ +1}$* to the updated values.

VSegType$_{TestIndex\ +1}$ = MACH CAS

Crossing Mach$_{TestIndex\ +1}$ = LastMach

Crossing $CAS_{TestIndex\ +1}$ *= MachToCas(LastMach,* $Altitude_{TestIndex}$*)*

$Mach_{i+1}$ *= LastMach*

CAS_{i+1} *= Crossing* $CAS_{TestIndex\ +1}$

Use a default crossing rate.

Crossing $Rate_{TestIndex\ +1}$ *= 0.25 kt/sec*

Mach Transition Altitude = $Altitude_{TestIndex\ +1}$

if (Valid = true) then

Add a TCP for the end of the TOD acceleration.

$d = DTG_{TestIndex}$ *- dx*

Find the ground track at this distance.

OldGndTrk = GetTrajGndTrk(d)

Save the wind data at this distance.

GenerateWptWindProfile(d, TemporaryWindProfile)

Find the position in the trajectory to insert the new TCP.

k = TestIndex + 1

fini = false

while (fini = false)

if ((k > index number of the first TCP) or (DTG_k *< d)) fini = true*

else k = k + 1

InsertTcp(k)

$TcpType_k$ *= VTCP*

$VSegType_k$ *= TOD DECELERATION*

$TurnType_k$ *= NO TURN*

DTG_k *= d*

$Altitude_k$ *=* $Altitude_{TestIndex}$ *- (NmiToFeet * dx) * tangent(Altitude Crossing Angle*$_{k+1}$*)*

Altitude Crossing Angle$_k$ *= Altitude Crossing Angle*$_{k+1}$

CAS_k = Mach Transition CAS

Crossing CAS_k = Mach Transition CAS

Mach Segment$_k$ = false

Use the default CAS rate.

Crossing Rate$_k$ = 0.25 kt/sec

if (WptInTurn(k)) Ground Track$_k$ = ComputeGndTrk(k, d)

else Ground Track$_k$ = OldGndTrk

Add the wind data to this new TCP.

Copy the wind data from TemporaryWindProfile to the wind data of TCP$_k$

end of if (Valid = true)

else mark this as a fatal error condition

Change TOD Mach Value

The *Change TOD Mach Value* function changes the TOD Mach value if the descent Mach, *Mach Descent Mach,* is different from the TOD Mach. This function is only invoked if the variable *MachCasAtTod* is false. The function also will add any required, additional TCPs.

The calling program provides as input and retains the subsequent outputs for the following variables: *TodId, TodMach, and TodMachRate*. The variable *TodId* is the name of the top-of-descent waypoint and is initialized as a null string by the calling program. Since this function may overwrite the Mach and speed change rate for an input waypoint, these variables allow the function to retain the original values for Mach and speed change rate and to then reset these variables to their original values prior to recalculating new values.

If the Mach value for the first TCP is not set, i.e., the path does not start with a Mach segment, or there is no defined descent Mach, i.e., *Mach Descent Mach = 0*, the function terminates. Otherwise,

If the previous TOD data for an input waypoint have been changed, these data are restored to their original values.

i = index number of the first TCP

The last designated Mach waypoint,

LastMachIndex = index number of the first TCP

The first designated CAS waypoint,

FirstCasIndex = index number of the first TCP

TodIndex = index number of the first TCP

Find the Mach and CAS waypoints.

fini = false

i = index number of the first TCP

while ((i ≤ index number of the last TCP) and (fini = false))

if (Crossing Mach$_i$ > 0) LastMachIndex = i

else if (Crossing CAS$_i$ > 0) then

FirstCasIndex = i

fini = true

i = i + 1

Find the TOD waypoint and Mach.

fini = false

i = index number of the first TCP

while ((i <index number of the last TCP) and (fini = false))

if ((Altitude$_i$ < Altitude$_{first\ TCP}$) or (Crossing CAS$_i$ > 0)) then

if (Altitude$_i$ ≠ Altitude$_{first\ TCP}$) TodIndex = i - 1

else TodIndex = i

fini = true

else if (Crossing Mach$_i$ > 0) MachAtTod = Crossing Mach$_i$

i = i + 1

If the vertical segment type has not been defined, mark this as the TOD.

if ((TodIndex > index number of the first TCP) and (VSegType$_{TodIdx}$ = NO TYPE))

VSegType$_{TodIdx}$ = TOD ALTITUDE

Check for errors. There cannot be a programmed descent Mach if there is a downstream Mach restriction.

if ((LastMachIndex > TodIndex) or (FirstCasIndex ≤ TodIndex)) mark this as a fatal error condition

else

Save the Mach values for all input waypoints so that they may be reset on subsequent passes back to their original input values.

if $(TcpType_{TodIndex} = INPUT)$ *then*

$TodId = Id_{TodIndex}$

$TodMach = Crossing\ Mach_{TodIndex}$

$TodMachRate = Crossing\ Rate_{TodIndex}$

if $((TcpType_{TodIndex} = INPUT)$ *and* $(Crossing\ Rate_{TodIndex} > 0))$

$CAS\ Rate = Crossing\ Rate_{TodIndex}$

else CAS Rate = 0.75 kt / sec (a default value)

The following is added to force a subsequent speed calculation.

$Crossing\ Rate_{TodIndex} = CAS\ Rate$

If the aircraft will slow during the descent, do the following:

if $(MachAtTod \geq Mach\ Descent\ Mach)$ *then*

Overwrite the TOD Mach value.

$Crossing\ Mach_{TodIndex} = Mach\ Descent\ Mach$

else

This is a special case where the aircraft is accelerating to the descent Mach.

Invoke the secondary function *DoTodAcceleration*. This function is described in a subsequent section.

DoTodAcceleration(TodIdx, MachAtTod)

$Crossing\ Mach_{TodIndex} = MachAtTod$

end of if $(MachAtTod \geq Mach\ Descent\ Mach)$

Compute Mach-to-CAS TCP

If a Mach-to-CAS transition is required, this function computes the Mach-to-CAS altitude and inserts a Mach-to-CAS TCP. This function is only performed if the input data starts with a Mach *Crossing Speed* for the first TCP. The function determines the appropriate Mach and CAS values, calculates the altitude that these values are equal, and then determines the along-path distance where this altitude occurs on the profile. Input into this function includes the variable *MachCasAtTod*. This variable is set in the function

Meet Cruise CAS Restriction and indicates that, if true, the Mach-to-CAS transitions occurs at the TOD point.

The following variables are initialized:

Mach Transition Altitude = 0

where this variable a part of the global path data.

The *Mach Segment* for each TCP is initialized to *false*.

for (i = index number of the first TCP; i ≤ index number of the last TCP; i = i + 1)

Mach Segment$_i$ = false

Other local variables are initialized.

fini = false

First CAS = 0

Last Mach = 0

CAS Constraint Flag = true

Mach Index = 0, where this variable is used to designate the last Mach waypoint.

Cas Index = -1, where this variable is used to designate the first CAS waypoint.

CAS Constraint Flag = true

If this is the special case where the TOD is the Mach-to-CAS transition point, insert the TCP here. This special case is determined in the function *Meet Cruise CAS Restriction.*

if (MachCasAtTod) then

Find the TOD.

i = index number of the first TCP

while ((i ≤ index number of the last TCP) and (fini = false))

if (VSegType$_i$ = TOD ALTITUDE) fini = true

else i = i + 1

InsertTcp(i+1)

Copy all of the data from TCP$_i$ into TCP$_{i+1}$

Now set the data in TCP$_{i+1}$ to the updated values.

$VSegType_{i+1}$ = MACH CAS

$Crossing\ Mach_{i+1}$ = Mach Descent Mach

$Crossing\ CAS_{i+1}$ = Mach Transition CAS

$Mach_{i+1}$ = Mach Descent Mach

CAS_{i+1} = Mach Transition CAS

Use the default CAS rate if the current rate is 0.

if ($Crossing\ Rate_{i+1} = 0$) $Crossing\ Rate_{i+1} = 0.25$ kt/sec

$Mach\ Transition\ Altitude = Altitude_{i+1}$

Set the Mach flag to true up to and including this point.

for (j = index number of the first TCP; $j <= i+1$; j++) $Mach\ Segment_j$ = true

end of if (MachCasAtTod)

else if ($Crossing\ Mach_{first\ TCP} > 0$) *then*

Perform the standard test for the Mach / CAS transition point.

CAS Constraint Flag = false

i = index number of the first TCP

while (($i <=$ index number of the last TCP) and (*fini = false*))

if ($Crossing\ Mach_i > 0$) *then*

$Last\ Mach = Crossing\ Mach_i$

$Mach\ Index = i$

else if ($Crossing\ CAS_i > 0$) *then*

$First\ CAS = Crossing\ CAS_i$

$CAS\ Rate = Crossing\ Rate_i$

$CAS\ Index = i$

CAS Constraint Flag = true

fini = true

$i = i + 1$

end of while

if (Mach Transition CAS > 0) First CAS = Mach Transition CAS

if (CAS Constraint Flag) then

z = FindMachCasTransitionAltitude(First CAS, Last Mach)

Determine if the very first TCP is already below the Mach-to-CAS transition altitude and z is greater or equal to 28,000 ft.

if ((Mach Index = 0) and (z > $Altitude_{first\ TCP}$) and (z >= 28000 ft)) then

Change the first TCP to CAS, using the descent CAS value if it is valid.

if (Mach Transition CAS > 0.) Crossing $CAS_{first\ TCP}$ = Mach Transition CAS

else Crossing $CAS_{first\ TCP}$ = First CAS

Set the entire speed profile to CAS.

fini = false

i = index number of the first TCP

while ((fini = false) and (i < (index number of the last TCP - 1)))

if (Crossing $Mach_i$ > 0) Crossing $Mach_i$ = 0

if (Crossing $CAS_i \neq 0$) fini = true

Mach Transition Altitude = z

Mach Transition CAS = 0

Mach Transition Mach = 0

end of if ((Mach Index = 0)...

Otherwise, determine if there is a Mach / CAS transition error.

else if ((z > $Altitude_{Mach\ Index}$) or (z < 18000 ft)) then

skip = false

Determine if the trajectory is already at a level altitude.

j = Mach Index

while ((j > index number of the first TCP) and ($TcpType_j \neq$ INPUT)) j = j - 1

if (Altitude$_j$ = Altitude$_{CAS\ Index}$) then

spd = MachToCas(Crossing Mach$_{Mach\ Index}$, Altitude$_j$)

if (spd ≥ Crossing CAS$_{CAS\ Index}$) then

Convert the Mach to a CAS crossing.

Crossing Mach$_j$ = Crossing Mach$_{Mach\ Index}$

Crossing CAS$_j$ = spd

Crossing Rate$_j$ = Crossing Rate$_{CAS\ Index}$

Crossing Altitude$_j$ = Altitude$_{CAS\ Index}$

if (Crossing Angle$_j$ = 0) then

if (Crossing Angle$_{CAS\ Index}$ ≠ 0) Crossing Angle$_j$ = Crossing Angle$_{CAS\ Index}$

else if (Crossing Angle$_{Mach\ Index}$ ≠ 0) Crossing Angle$_j$ = Crossing Angle$_{Mach\ Index}$

else Crossing Angle$_j$ = 2.4 degrees

end if (Crossing Angle$_j$ = 0)

VSegType$_j$ = MACH CAS

Mach$_j$ = Last Mach

CAS$_j$ = spd

Mach Transition Altitude = Altitude$_j$

Mach Transition CAS = spd

for (k = index number of the last TCP; k ≤ j; k++) Mach Segment$_k$ = true

skip = true

end of if (spd >= Crossing CAS$_{CAS\ Index}$)

end of if (Altitudej = Altitude$_{CAS\ Index}$)

if (skip = false) Set an error indicating a bad Mach-to-CAS transition.

end of else if ((z > Altitude$_{Mach\ Index}$)...

else

i = index of the first TCP + 1

fini = false

while ((i < index of the last TCP) and (fini = false))

if (Altitude$_i$ > z) i = i + 1

else fini = true

Calculate the distance to *Altitude$_i$*.

z2 = Altitude$_{i-1}$ - Altitude$_i$

if (z2 <= 0) rz = 0

else rz = (z - Altitude$_i$) / z2

*d = rz * (DTG$_{i-1}$ - DTG$_i$) + DTG$_i$*

GndTrk = GetTrajGndTrk(d)

Add the new TCP.

InsertTcp(i)

TcpType$_i$ = VTCP

VSegType$_i$ = MACH CAS

TurnType$_i$ = NO TURN

Crossing Mach$_i$ = Last Mach

Crossing CAS$_i$ = First CAS

Crossing Rate$_i$ = CAS Rate

DTG$_i$ = d

Altitude$_i$ = z

Crossing Angle$_i$ =Altitude Crossing Angle$_{i+1}$

Ground Track$_i$ = GndTrk

Mach$_i$ = Last Mach

CAS$_i$ = First CAS

Mach Transition Altitude = z

Mach Transition CAS = First CAS

Compute and add the wind data at distance d along the path to the data of TCP_i.

GenerateWptWindProfile(DTG_i, TCP_i)

Set the Mach flag for these TCPs.

for (j = index number of the first TCP; j < i; j++) Mach Segment$_j$ = true

end of else

end of if (CAS Constraint Flag)

else

There are only Mach segments, set the Mach flags to true.

for (j = index number of the first TCP; j < index number of the last TCP; j++)
Mach Segment$_j$ = true

if ((Mach Transition Mach = 0) and (Crossing Mach$_{last\ TCP}$ > 0))

Mach Transition Mach = Crossing Mach$_{last\ TCP}$

end of else if (Crossing Mach$_{first\ TCP}$ > 0)

Compute Altitude / CAS Restriction TCP

If an altitude / CAS restriction is required, the *Compute Altitude / CAS Restriction TCP* function computes the altitude / CAS restriction point and inserts an altitude / CAS TCP. This is the (U.S.) point where the trajectory transitions through 10,000 ft and a 250 kt restriction is required. This function is only performed if the previously computed flag *Need10KRestriction* is true. The function determines the along-path distance where this altitude / CAS restriction occurs on the profile. A TCP is then inserted into the TCP list at this point. The restriction values are *Descent Crossing Altitude* and *Descent Crossing CAS*.

Find the first TCP that is below the *Descent Crossing Altitude* in the list.

i = index number of the first TCP

k = i

fini = false

while ((i <index number of the last TCP) and (fini = false))

if (Altitude$_i$ < Descent Crossing Altitude) then

k = i

fini = true

i = i + 1

Find the last CAS restriction prior to the first TCP below *Descent Crossing Altitude*.

i = k - 1

fini = false

Last CAS = 0

while ((i > index number of the first TCP) and (fini = false))

if (Crossing CAS$_i$ > 0) then

Last CAS = Crossing CAS$_i$

fini = true

i = i - 1

Determine if an altitude or CAS TCP is required. If it is, add it.

if ((Mach Segment$_k$ = true) and (Last CAS > Descent Crossing CAS)) then

A crossing restriction needs to be added.

i = k

Find the distance to this altitude.

x = Altitude$_{i-1}$ - Altitude$_i$

if (x ≤ 0) ratio = 0

else ratio = (Descent Crossing Altitude - Altitude$_i$) / x

*d = ratio * (DTG$_{i-1}$ - DTG$_i$) + DTG$_i$*

Compute the ground track at distance *d* along the trajectory and save it as *Saved Ground Track*.

Saved Ground Track = GetTrajGndTrk(d)

Insert a new TCP at location *i* in the TCP list. The TCP is inserted between TCP$_{i-1}$ and TCP$_i$ from the original list. The function *InsertTcp* should be appropriate for the actual data structure implementation of this function.

InsertTcp(i)

Mark this TCP as the altitude / CAS restriction TCP.

TcpType$_i$ = VTCP

if (VSegType$_i$ = no type) VSegType$_i$ = ALTITUDE CAS RESTRICTION

$TurnType_i$ = *NO TURN*

Add the data for this new TCP.

$Crossing\ Mach_i$ = *0*

$Crossing\ CAS_i$ = *Descent Crossing CAS*

Use a high value, arbitrary CAS rate.

$CAS\ Rate_i$ = *0.75 kt / sec*

DTG_i = *d*

$Altitude_i$ = *Descent Crossing Altitude*

$Crossing\ Angle_i$ = $Crossing\ Angle_{i+1}$

Set the Mach flag for TCP_i *to false*

$Ground\ Track_i$ = *Saved Ground Track*

$Mach_i$ = *0*

CAS_i = *Descent Crossing CAS*

Compute and add the wind data at distance *d* along the path to the data of TCP_i.

GenerateWptWindProfile(DTG_i, TCP_i*)*

Add Final Deceleration

The Add Final deceleration function generates the appropriate speed TCP's for the case where either the deceleration to the final approach speed is to begin at the Final Approach Fix or the deceleration is to end at a specific altitude, *Stable Altitude*. This latter option is to support the case, which is typical for air transport operations, where a stable approach is required at and below a specific altitude. This function may only be invoked if the last TCP is the runway threshold and the input crossing speed is a valid CAS value.

if ((Final Deceleration Option = AT FAF) or (Final Deceleration Option = STABLE)) then

The runway waypoint.

RunwayWpt = index number of the last TCP

The speed specified at the last TCP, which must be the runway, is the target speed for these options. This speed should be the corrected final approach speed, *CFAS*.

CFAS = $Crossing\ CAS_{last\ TCP}$

Find the waypoint index number for the waypoint used as the FAF. The default value is the input waypoint just before the last TCP. If there exists a FAF waypoint named in the input data, *NamedFaf*, then use that waypoint.

FafWpt = RunwayWpt - 1

if (NamedFaf) then

Find this waypoint by name.

found = false

k = FafWpt

while ((found = false) and (k > index number of the first TCP))

if (NamedFaf = Id_k) found = true

else k = k - 1

if (found) FafWpt = k

The following is for the deceleration at the FAF.

if (Final Deceleration Option = AT FAF) then

delta = Crossing CAS_{FafWpt} - CFAS

Find the time required to reach the final speed.

t = delta / Crossing $Rate_{RunwayWpt}$ / (3600 sec/hr)

Find the FAF altitude.

if (Crossing $Altitude_{FafWpt}$ > 0)

AltitudeFaf = Crossing $Altitude_{FafWpt}$

else if (Crossing $Angle_{RunwayWpt}$ ≤ 0)

There is no way to accurately calculate the altitude, use the runway altitude.

AltitudeFaf = Crossing $Altitude_{RunwayWpt}$

else

AltitudeFaf = Crossing $Altitude_{RunwayWpt}$ +
*(DTG_{FafWpt} * NmiToFeet) * tangent(Crossing $Angle_{RunwayWpt}$)*

Calculate the ground speed at the runway.

InterpolateWindWptAltitude(Wind $Profile_{RunwayWpt}$, $Altitude_{RunwayWpt}$, Ws, Wd, Td)

GsRny = ComputeGndSpeedUsingTrack (Crossing $CAS_{RunwayWpt}$, $GndTrack_{RunwayWpt}$, $Altitude_{RunwayWpt}$, Ws, Wd, Td)

Calculate the ground speed at the FAF.

InterpolateWindWptAltitude(Wind $Profile_{FafWpt}$, $Altitude_{FafWpt}$, Ws, Wd, Td)

GsFaf = ComputeGndSpeedUsingTrack (Crossing CAS_{FafWpt},$GndTrack_{FafWpt}$, $Altitude_{FafWpt}$,Ws,Wd,Td)

Calculate the distance from the FAF toward the runway where the final speed will be reached.

*x = (GsFaf + GsRny) / 2 * t*

Calculate the distance from the runway.

dtg = DTG_{FafWpt} - x

Now find this distance in the TCP's.

TmpWpt = RunwayWpt

while ((DTG_{TmpWpt} < dtg) and (TmpWpt > index number of the first TCP))

TmpWpt = TmpWpt - 1

Now find the next downstream input waypoint.

while (($TcpType_{TmpWpt}$ ≠ INPUT) and (TmpWpt < RunwayWpt))

TmpWpt = TmpWpt + 1

GndTrk2 = $GndTrack_{TmpWpt}$

Using the just computed estimates, recalculate the DTG.

if (Crossing $Angle_{RunwayWpt}$ ≤ 0) Delta Z = 0

*else Delta Z = (x * NmiToFeet) * tangent(Crossing $Angle_{RunwayWpt}$)*

Altitude2 = AltitudeFaf - Delta Z

Find the wind value between the two points.

InterpolateWindWptAltitude(Wind $Profile_{FafWpt}$, Altitude2, Spd0, Dir0, TDev0)

InterpolateWindWptAltitude(Wind $Profile_{TmpWpt}$, Altitude2, Spd1, Dir1, TDev1)

if (dtg > 0) InterpolateWindAtRange(dtg, DTG_{FafWpt}, Spd0, Dir0, TDev0, 0, Spd1, Dir1, TDev1, WindSpd, WindDir, TempDev)

else

WindSpd = Spd1

WindDir = Dir1

TempDev = TDev1

Calculate the ground speed at the deceleration point.

DecelGs = ComputeGndSpeedUsingTrack(CFAS, GndTrk2, Altitude2, WindSpd, WindDir, TempDev)

Calculate the average ground speed.

AvgGs = (GsFaf + DecelGs) / 2

Calculate the distance for the speed change.

*x = AvgGs * t*

Calculate the distance from the runway for this speed point.

dtg = DTG_{FafWpt} - x

end of if (Final Deceleration Option = AT FAF)

else

Calculate the data for the stabilized altitude option.

StableAlt = $Crossing\ Altitude_{RunwayWpt}$ + Stable Altitude

dtg = (Stable Altitude / NmiToFeet) / tangent($Crossing\ Altitude_{RunwayWpt}$)

Find the waypoint prior to the stable altitude.

TmpWpt = RunwayWpt

while ((DTG_{TmpWpt} < dtg) and (TmpWpt > index number of the first TCP))

TmpWpt = TmpWpt - 1

Save the ground track at this point.

GndTrk2 = $Ground\ Track_{TmpWpt}$

Calculate the wind data at the two positions.

InterpolateWindWptAltitude($Wind\ Profile_{FAFWpt}$, StableAlt, Spd0, Dir0, TDev0)

InterpolateWindWptAltitude($Wind\ Profile_{TmpWpt}$, StableAlt, Spd1, Dir1, TDev1)

Interpolate the winds between the two waypoints.

if (dtg > 0) InterpolateWindAtRange(dtg, DTG_{FafWpt}, Spd0, Dir0, TDev0,
0, Spd1, Dir1, TDev1, WindSpd, WindDir, TempDev)

else

WindSpd = Spd1

WindDir = Dir1

TempDev = TDev1

Calculate the ground speed at the deceleration point.

DecelGs = ComputeGndSpeedUsingTrack(CFAS, GndTrk2, StableAlt, WindSpd,
WindDir, TempDev)

end of else { Calculate the data for the stabilized altitude option }

Add the appropriate speed TCP if its position is between the FAF and the runway and the CFAS is slower than the speed at the FAF.

if ((dtg > 0) and (dtg ≤ DTG_{FafWpt}) and (Crossing CAS_{FafWpt} > CFAS)) then

Save the original ground track value at this distance.

GndTrk = GetTrajGndTrk(dtg)

Find the position in the TCP list to insert this waypoint.

i = RunwayWpt

while ((DTG_i < dtg) and (i > index number of the first TCP)) i = i - 1

Define the correct insertion point.

i = i + 1

InsertTcp(i)

$TcpType_i$ = VTCP

if ($VSegType_i$ = NO TYPE) $VSegType_i$ = FINAL SPEED

$TurnType_i$ = NO TURN

Crossing $Mach_i$ = 0.

Crossing CAS_i = Crossing $CAS_{RunwayWpt}$

$Crossing\ Rate_i = Crossing\ Rate_{RunwayWpt}$

$DTG_i = dtg$

Calculate the altitude at this point.

if $((DTG_{i-1} - DTG_{i+1}) \leq 0)$ $x = 0$

else $x = (DTG_i - DTG_{i+1}) / (DTG_{i-1} - DTG_{i+1})$

$Altitude_i = x * Altitude_{i-1} + (1 - x) * Altitude_{i+1}$

$Mach\ Segment_i = false$

$Crossing\ Angle_i = Crossing\ Angle_{i+1}$

$Ground\ Track_i = GndTrk$

$Ground\ Speed_i = DecelGs$

$Mach_i = 0$

$CAS_i = Crossing\ CAS_i$

Compute and add the wind data at the new TCP's DTG.

GenerateWptWindProfile(DTG_i, TCP_i)

end of adding the TCP

else mark this as a fatal error condition

end of if ((Final Deceleration Option = AT FAF) or (Final Deceleration Option = STABLE))

Add Waypoint at 6.25 nmi

The Add Waypoint at 6.25 nmi function generates a special waypoint at 6.25 nmi before the landing threshold of the runway. This function is invoked if the input variable *AddMopsRWY625* is true. This capability to support this special waypoint at 6.25 nmi before the threshold, along with associated crossing altitude and speed conditions, is a requirement of the RTCA *Minimum Operational Performance Standards (MOPS) for Flight-deck Interval Management (FIM)* (ref. 34). This function may only be invoked if the last TCP is the runway threshold and the input crossing speed is a valid CAS value.

if (AddMopsRWY625 and $(Crossing\ CAS_{last\ TCP} > 0)$*) then*

error = false

LastNum = index number of the last TCP

Determine where the 6.25 nmi needs to be placed in the TCP list.

found = false

i1 = LastNum

while ((found = false) and (i1 > index number of the first TCP))

Find the named waypoint at-or-before 6.25 nm in the TCP records.

if ((TcpType$_{i-1}$ = INPUT) and (DTG$_{i-1}$ > 6.25 nmi)) found = true

i1 = i1 - 1

if (found = false) error = true

Find the upstream waypoint with a speed constraint.

j = i1

found2 = false

while ((found2 = false) and (j ≥ index number of the first TCP))

if ((TcpType$_j$ = INPUT) and (Crossing CAS$_j$ > 0)) found2 = true

else j = j - 1

if (found2 = false) error = true

spd = Crossing CAS$_j$

The MOPS requires that the crossing speed cannot be faster than 170 kt.

if (spd > 170 kt) spd = 170 kt

Find the downstream CAS rate.

j = i1 + 1

found2 = false

while ((found2 = false) and (j ≤ index number of the last TCP))

if ((TcpType$_j$ = INPUT) and (Crossing CAS$_j$ > 0)) found2 = true

else j = j + 1

if (found2 = false) error = true

spdrate = Crossing Rate$_j$

Set the rate to a minimum of 0.75 kt / sec.

if (spdrate < 0.75 kt /sec) spdrate = 0.75 kt / sec

Find the downstream descent data.

j = i1 + 1

found2 = false

while ((found2 = false) and (j < index number of the last TCP))

if ((TcpType$_j$ = INPUT) and (Crossing Altitude$_j$ > 0)) found2 = true

else j = j + 1

if (found2 = false) error = true

This point needs to be crossed at an altitude of at least 2000 ft above the runway altitude.

alt = Crossing Altitude$_{last\ TCP}$ + 2000 ft

if (alt ≤ Crossing Altitude$_j$) then

alt = Crossing Altitude$_j$

angle = Crossing Angle$_j$

else

angle = Crossing Angle$_j$

if (angle < Crossing Angle$_{last\ TCP}$) angle = Crossing Angle$_{last\ TCP}$

Check the actual calculated altitude.

z = alt - Crossing Altitude$_j$

if (z > 0) then

d = 6.25 nmi - DTG$_j$

if (d > 0) then

*a = arctangent(z, NmiToFeet * d)*

if (a > angle) angle = a

Find the waypoint after this in the input waypoint data.

found2 = false

j1 = index number of the last TCP

while ((found = false) and (j1 ≥ index number of the first TCP))

if (Id_{j1} = Id_{i1}) found2 = true

else j1 = j1 - 1

if (found = false) error = true

Find the next named waypoint after 6.25 nm in the input data.

j0 = j1

found2 = false

i0 = index number of the last TCP

while ((found2 = false) and (i0 ≥ index number of the first TCP))

if (($TcpType_{i0}$ = INPUT) and (Id_{j0} = Id_{i0})) found2 = true

else i0 = i0 - 1

if (found2 = false) error = true

If there are no errors, insert the 6.25 nmi point.

if (error= false) then

GndTrk = GetTrajGndTrk(6.25 nmi)

Find the position to insert this waypoint.

i = index number of the last TCP

while ((DTG_i < 6.25 nmi) and (i > index number of the first TCP)) i = i - 1

The correct insertion point is the next downstream point.

i = i + 1

InsertTcp(i)

$TcpType_i$ = VTCP

$VSegType_i$ = RUNWAY625

$TurnType_i$ = NO TURN

Crossing $Mach_i$ = 0

Crossing CAS_i = spd

Crossing $Rate_i$ = spdrate

$DTG_i = 6.25$ nmi

$Altitude_i = alt$

$Crossing\ Altitude_i = alt$

$Mach\ Segment_i = false$

$Crossing\ Angle_i = angle$

$Ground\ Track_i = GndTrk$

$Mach_i = 0$

$CAS_i = Crossing\ CAS_i$

Add the wind data at this distance.

$GenerateWptWindProfile(DTG_i, TCP_i)$

$InterpolateWindWptAltitude(Wind\ Profile_i, Crossing\ Altitude_i, WindSpd, WindDir, TempDev)$

$Ground\ Speed_i = ComputeGndSpeedUsingTrack(Crossing\ CAS_i, Ground\ Track_i,$
$Crossing\ Altitude_i, WindSpd, WindDir, TempDev)$

If there is a programmed deceleration at the original FAF and the FAF is farther from the runway than 6.25 nmi, remove the previously computed final deceleration point.

if ((Final Deceleration Option = AT FAF) or (Final Deceleration Option = STABLE)) then

Find the index number for the FAF. Initialize the index to an invalid number, -1.

FafWptNum = -1

Is this the special case with a named FAF, *NamedFaf,* in the input?

if (NamedFaf) then

Find this waypoint by name.

found = false

k = index number of the last TCP

while ((found = false) and (k > index number of the first TCP))

if (NamedFaf = Id_k) then

found = true

FafWptNum = k

else k = k - 1

else

FafWptNum = index number of the last TCP - 1

while ((FafWptNum > index number of the first TCP) and
($TcpType_{FafWptNum} \neq INPUT$))

FafWptNum = FafWptNum - 1

found2 = false

i = index number of the last TCP

while ((found2 = false) and (FafWptNum > index number of the first TCP) and
(i > index number of the first TCP))

if ($VSegType_i$ = FINAL SPEED) found2 = true

else i = i - 1

if ((found2 = true) and ($DTG_{FafWptNum}$ > 6.25 nmi)) RemoveWaypoint(i)

where the RemoveWaypoint function simply deletes the TCP at the index *i*.

end of if (error= false)

else mark this as a fatal error condition

Compute TCP Speeds

The *Compute TCP Speeds* function is similar to *Compute TCP Altitudes* in its design. Beginning with the last waypoint, this function computes the Mach or CAS at each previous TCP and inserts any additional speed TCPs that may be required to denote a change in the speed profile. The function uses the current speed constraint, searches backward for the previous constraint, and then computes the distance required to meet this previous constraint. The speeds for all of the TCPs within this distance are computed and added to the data for the TCPs. If the along-path distance to meet the previous constraint is not at a TCP, a new speed VTCP is inserted at this distance. This function invokes two secondary functions, described in the subsequent text, with the invocation dependent on the constraint speed, whether it is a Mach or a CAS value. This function is performed in the following steps:

The speed of the first TCP is set to its crossing speed.

if (Crossing $Mach_{first\ TCP}$ > 0) then

$Mach_{first\ TCP}$ = Crossing $Mach_{first\ TCP}$

$CAS_{first\ TCP}$ = MachToCas($Mach_{first\ TCP}$, $Altitude_{first\ TCP}$)

else

$CAS_{first\ TCP} = Crossing\ CAS_{first\ TCP}$

$Mach_{first\ TCP} = CasToMach(CAS_{first\ TCP},\ Altitude_{first\ TCP})$

Set the current constraint index number, *cc*, equal to the index number of the last TCP, which is typically the runway threshold,

cc = index number of the last TCP

A flag signifying that Mach segment computation has begun is initially set to false,

Doing Mach = false

Check for special case where there are no CAS segments.

if ((Crossing $CAS_{cc} = 0$*) and (Crossing* $Mach_{cc} > 0.0$*)) then*

$CAS_{cc} = MachToCas(Crossing\ Mach_{cc},\ Crossing\ Altitude_{cc})$

$Mach_{cc} = Crossing\ Mach_{cc}$

DoingMach = true

else $CAS_{cc} = Crossing\ CAS_{cc}$

while (cc > index number of the first TCP)

Set the Mach flag if the current TCP is the Mach-to-CAS transition point.

$if\ (TCP_{cc} = Mach\ Transition\ CAS)\ Doing\ Mach = true$

if (Doing Mach) ComputeTcpMach(cc)

else ComputeTcpCas(cc)

end of while cc > index number of the first TCP

Compute Secondary Speeds

The *Compute Secondary Speeds* function adds the Mach values to CAS TCPs, the CAS values to Mach TCPs, and the ground speed values to all TCPs. This function is performed in the following steps:

Doing Mach = false

If the last TCP input speed is defined as Mach, set the Mach flag to true.

$if\ (Crossing\ Mach_{last\ TCP} > 0)\ \ DoingMach = true$

Working backwards from the runway, compute the relevant speeds.

for (i = index number of the last TCP; i ≥ index number of the first TCP; i = i - 1)

Set the flag if the current TCP is the Mach-to-CAS transition point.

if (VSegType$_i$ = MACH CAS) Doing Mach = true

if (Doing Mach) Cas$_i$ = MachToCas(Mach$_i$, Altitude$_i$)

else Mach$_i$ = CasToMach(Cas$_i$, Altitude$_i$)

Compute the ground track.

if (i = index number of the first TCP) track = Ground Track$_i$

else if (WptInTurn(i) or (TcpType$_i$ = END TURN)) track = Ground Track$_i$

else track = Ground Track$_{i-1}$

Compute the ground speed. This also requires the computation of the wind at this point.

InterpolateWindWptAltitude(Wind Profile$_i$, Altitude$_i$,Wind Speed, Wind Direction, Temperature Deviation)

Ground Speed$_i$ = ComputeGndSpeedUsingTrack (Cas$_i$, track, Altitude$_i$, Wind Speed, Wind Direction, Temperature Deviation)

end of for (i = index number of the last TCP; i ≥ index number of the first TCP; i = i - 1)

Compute Turn Data

The *Compute Turn Data* function computes the turn data for each turn waypoint and modifies the associated waypoint's turn data sub-record. This function performs as follows:

KtsToFps = 1.69

Nominal Bank Angle = 22°

index = index number of the first TCP + 1

while (index < index number of the last TCP)

Find the next input waypoint with a turn.

while ((index < index number of the last TCP) and ((TcpType$_{index}$ ≠ INPUT) or (WptInTurn(index) = false))) index = index + 1

If there are no errors and there is a turn of more than 3-degrees, compute the turn data.

if (index < index number of the last TCP) then

Find the start of the turn.

$i = index - 1$

$while\ (TcpType_i \neq BEGIN\ TURN)\ i = i - 1$

$start = i$

The following are all approximations and are based on a general, constant radius turn.

The start of turn to the midpoint data is as follows, noting that the ground speeds for all points must be valid at this point.

The overall distance d for this part of the turn is,

$d = DTG_{start} - DTG_{index}$

The special case with 0 distance between the points is,

$if\ (d \leq 0)\ AvgGsFirstHalf = (Ground\ Speed_{start} + Ground\ Speed_{index}) / 2$

else

The overall average ground speed is computed as follows, noting that it is the sum of segment distance / overall distance * average segment ground speed.

$AvgGsFirstHalf = 0$

$for\ (j = start;\ j \leq (index - 1);\ j = j + 1)$

$dx = DTG_j - DTG_{j+1}$

$AvgGsFirstHalf = AvgGsFirstHalf + (dx / d) * (Ground\ Speed_j + Ground\ Speed_{j+1}) / 2$

Now, find the end of the turn.

$i = index + 1$

$while\ (TcpType_i \neq END\ TURN)\ i = i + 1$

$end = i$

Now, find the midpoint to the end of the turn.

The overall distance for this part of the turn is,

$d = DTG_{index} - DTG_{end}$

Test for the special case, 0 distance between the points.

$if\ (d \leq 0)\ AvgGsLastHalf = (Ground\ Speed_{index} + Ground\ Speed_{end}) / 2$

else

Compute the overall average ground speed noting that it is the sum of the segment distances / overall distance * average segment ground speed.

AvgGsLastHalf = 0

for (j = index; j ≤ (end - 1); j = j+ 1)

dx = DTG_j *-* DTG_{j+1}

*AvgGsLastHalf = AvgGsLastHalf + (dx / d) * (Ground Speed*$_j$ *+ Ground Speed*$_{j+1}$*) / 2*

end of for (j = index; j ≤ (end - 1); j = j + 1)

end of else if (d ≤ 0)

full turn = DeltaAngle(Ground Track$_{start}$*, Ground Track*$_{end}$*)*

half turn = full turn / 2

Compute the outputs from the average ground speed values.

Average Ground Speed = (AvgGsFirstHalf + AvgGsLastHalf) / 2

Save the ground speed data in the turn data for this waypoint.

Turn Data Average Ground Speed$_{index}$ *= Average Ground Speed*

Compute the turn radius and associated data. This set of calculations is not performed if the waypoint is a special, RF center-of-turn turn waypoint.

if (TurnType$_i$ *≠ RF TURN CENTER) then*

The general equation is turn rate = c tan(bank angle) / v. If the bank angle is a constant, turn rate = c0 / v. The *Nominal Bank Angle* = 22 degrees.

*c0 = 57.3 * 32.2 / KtsToFps * tangent(Nominal Bank Angle)*

Test for a negative ground speed.

if (Average Ground Speed ≤ 0) then

Turn Data Turn Time$_{index}$ *= 0*

Turn Data Turn Radius$_{index}$ *= 0*

else

w = c0 / Average Ground Speed

The time to make the turn is,

$Turn\ Data\ Turn\ Time_{index} = |full\ turn| / w$

The turn radius is,

$Turn\ Data\ Turn\ Radius_{index} =$
$(57.3 * KtsToFps * Average\ Ground\ Speed) / (NmiToFeet * w)$

The along-path distance for the turn is,

$Turn\ Data\ Path\ Distance_{index} = |full\ turn| * Turn\ Data\ Turn\ Radius_{index} / 57.3$

end of if ($TurnType_i \neq$ RF TURN CENTER)

else

These are the data for an RF turn. The along-path distance for the turn is,

$Turn\ Data\ Path\ Distance_{index} = |full\ turn| * Turn\ Data\ Turn\ Radius_{index} / 57.3$

Calculate the time to make the turn.

Test for a negative ground speed.

if (Average Ground Speed ≤ 0) $Turn\ Data\ Turn\ Time_{index} = = 0$

else

$Turn\ Data\ Turn\ Time_{index} =$
$(3600\ sec/hr) * Turn\ Data\ Path\ Distance_{index} / Average\ Ground\ Speed$

end of else if ($TurnType_i \neq$ RF TURN CENTER)

Save the turn data for the first half of the turn, denoted by the "1" in the variable name.

$Turn\ Data\ Cas1_{index} = CAS_{start}$

$Turn\ Data\ Average\ Ground\ Speed1_{index} = AvgGsFirstHalf$

$Turn\ Data\ Track1_{index} = Ground\ Track_{start}$

The *Straight Distance* values are the distances from the turn-entry TCP to the waypoint and from the waypoint to the turn-exit TCP. See the example in figure 10.

$Turn\ Data\ Straight\ Distance1_{index} = Turn\ Data\ Turn\ Radius_{index} * tangent(|half\ turn|)$

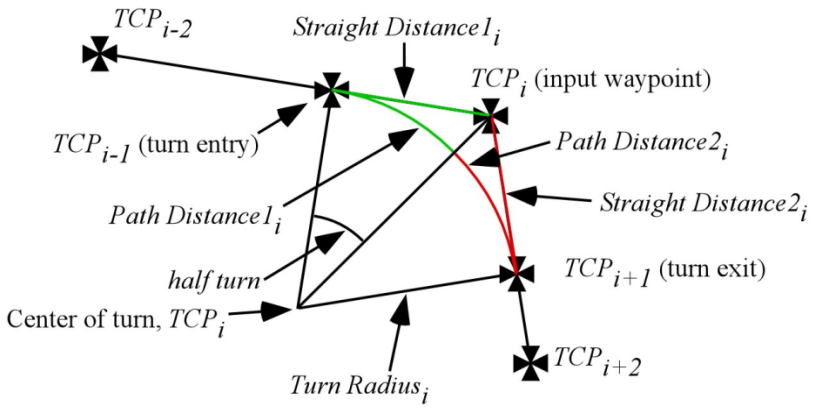


Figure 10. Turn distances for waypoint$_i$.

The Path Distance values are the along-the-path distances from the turn-entry TCP to a point one-half way along the turn and from this point to the turn-exit TCP. See the example in figure 10.

Turn Data Path Distance1$_{index}$ = |*half turn*| * *Turn Data Turn Radius*$_{index}$ / *57.3*

Compute the midpoint waypoint data. This set of calculations is not performed if the waypoint is a special, RF center-of-turn waypoint.

if (TurnType$_i \neq$ *RF TURN CENTER) then*

Test for a negative ground speed.

if (AvgGsFirstHalf $\leq$ *0) Turn Data Turn Time1*$_{index}$ = *0*

else

w = c0 / AvgGsFirstHalf

Turn Data Turn Time1$_{index}$ = |*half turn*| / *w*

else

These are the data for an RF turn.

Turn Data Turn Time1$_{index}$ = *Turn Data Path Distance1*$_{index}$ / *AvgGsFirstHalf * (3600 sec/hr)*

The data for the midpoint to the end of the turn, denoted by the "2" in the variable name, are as follows:

Turn Data Cas2$_{index}$ = *CAS*$_{end}$

Turn Data Average Ground Speed2$_{index}$ = *AvgGsLastHalf*

Turn Data Track2$_{index}$ = *Ground Track*$_{end}$

The distances for the second half of the turn are the same as for the first, but their calculations are recomputed here for clarity.

Turn Data Straight Distance2$_{index}$ *= Turn Data Turn Radius* $_{index}$ ** tangent(|half turn|)*

Turn Data Path Distance2$_{index}$ *= |half turn| * Turn Data Turn Radius*$_{index}$ */ 57.3*

Compute the data for the last half of the turn. Again, this set of calculations is not performed if the waypoint is a special, RF center-of-turn waypoint.

if (TurnType$_i$ *≠ RF TURN CENTER) then*

Test for a negative ground speed.

if (AvgGsFirstHalf ≤ 0) Turn Data Turn Time2$_{index}$ *= 0*

else

w = c0 / AvgGsLastHalf

Turn Data Turn Time2$_{index}$ *= |half turn| / w*

else

These are the data for an RF turn.

Turn Data Turn Time2$_{index}$ *= Turn Data Path Distance2*$_{index}$ */ AvgGsLastHalf * (3600 sec/hr)*

The *DTG* values are as follows:

DTG$_{start}$ *= DTG*$_{index}$ *+ Turn Data Path Distance1*$_{index}$

DTG$_{end}$ *= DTG*$_{index}$ *- Turn Data Path Distance2*$_{index}$

Since the turn waypoints have been moved, the wind data need to be updated for the new locations.

if (TcpType$_{start}$ *≠ INPUT) GenerateWptWindProfile(DTG*$_{start}$*, TCP*$_{start}$*)*

if (TcpType$_{end}$ *≠ INPUT) GenerateWptWindProfile(DTG*$_{end}$*, TCP*$_{end}$*)*

end of if (index < index number of the last TCP)

index = index + 1

end of while (index < index number of the last TCP)

Test for Altitude / CAS Restriction Requirement

The *Test for Altitude / CAS Restriction Requirement* function determines if the addition of an altitude / CAS restriction point is required. This is the (U.S.) point where the trajectory transitions through 10,000 ft

and a 250 kt restriction is required. This function determines the value of the *Need10KRestriction* flag. The function can only be called after an initial, preliminary trajectory has been generated. The restriction values are *Descent Crossing Altitude* and *Descent Crossing CAS*. If this function determines that the restriction is required, then the function *Compute Altitude / CAS Restriction TCP* will generate the restriction TCP.

Need10KRestriction = false

if ((Descent Crossing Altitude > 0) and (Descent Crossing CAS > 0)) ok = true

else ok = false

If the path does not start above 10,000ft, skip this routine.

if (ok and ($Altitude_{first\ TCP}$ > Descent Crossing Altitude)) then

Find the first point below *Descent Crossing Altitude*

fini = false

i = index number of the first TCP

while ((i <index number of the last TCP) and (fini = false))

if ($Altitude_i$ < Crossing Altitude) then

Find the distance to this altitude.

x = $Altitude_{i-1}$ - $Altitude_i$

if (x ≤ 0) ratio = 0

else ratio = (Descent Crossing Altitude - $Altitude_i$) / x

*s = ratio * (CAS_{i-1} - CAS_i) + CAS_i*

if (s > (Descent Crossing Cas + 2 kt)) Need10KRestriction = true

fini = true

i = i + 1

Update DTG Data

The *Update DTG Data* function is performed after the turn data have been updated and the VTCPs have been deleted. Only input, turn-entry, and turn-exit TCPs should be in the list at this time. If the input test flag, *TestOnly*, is true, then only the testing portions of this function are used.

if (TestOnly = false) $DTG_{first\ TCP}$ = 0

i = index number of the last TCP

while (i > index number of the first TCP)

Determine if there is a turn at either end and adjust accordingly.

if (WptInTurn(i))

if (TestOnly = false) $DTG_{i-1} = DTG_i$ + *Turn Data Path Distance1*$_i$

The following is the difference between going directly from the waypoint to going along the curved path.

PriorDistanceOffset = Turn Data Straight Distance1$_i$ - *Turn Data Path Distance1*$_i$

else PriorDistanceOffset = 0

Find the next input waypoint.

$n = i - 1$

while (TcpType$_n \neq$ *INPUT)* $n = n - 1$

if (WptInTurn(n))

The following is the difference between going directly from the waypoint to going along the curved path.

DistanceOffset = Turn Data Straight Distance2$_n$ - *TurnData.PathDistance2*$_n$

The DTG to the input waypoint is then:

if (TestOnly = false) DTG_n = *Center to Center Distance*$_i$ - *PriorDistanceOffset - DistanceOffset* + DTG_i

If the *DistanceOffset* is greater than *Center to Center Distance*$_i$, then the turn is too big.

if (DistanceOffset > Center to Center Distance$_i$*) mark this as a fatal error condition*

The turn-exit DTG is then,

if (TestOnly = false) $DTG_{n+1} = DTG_n$ - *Turn Data Path Distance2*$_n$

else if (TestOnly = false) then

The next waypoint is not in a turn.

DTG_n = *Center to Center Distance*$_i$ - *PriorDistanceOffset* + DTG_i

$i = n$

end of while (i > index number of the first TCP)

Find Linear Deceleration Segment DTG

This routine is used in the linear deceleration rate calculations to find the prior waypoint with a speed constraint and, if the current trajectory is valid, find the DTG to this waypoint.

Find the next linear deceleration rate waypoint whose speed has not been calculated using the secondary function *FindNextLDRWaypoint*, described in a subsequent section.

idx = FindNextLDRWaypoint()

if (idx > -1) then

$LDR\ Obtained\ Distance_{idx} = -1$

Find the previous speed constraint using the secondary function *LDRFindLastSpeedConstraint*, described in a subsequent section.

lastidx = LDRFindLastSpeedConstraint(idx)

$LDR\ Base\ Distance_{idx} = DTGlast_{idx}$

if $((Mach\ Segment_{idx} = false)$ *and* $(Mach\ Segment_{lastidx} = true))$ *then*

fini = false

j = idx

while $(j \geq$ *index number of first TCP) and (fini = false))*

if $(Mach\ Segment_j = true)$ *fini = true*

else j = j - 1

$lastspd = Cas_j$

else if (Mach Segmentidx = true) $lastspd = Mach_{lastidx}$

else $lastspd = Cas_{lastidx}$

Now find where the previous speed constraint is obtained.

idx2 = LDRFindAcquiredSpeed(idx,lastspd)

$LDR\ Obtained\ Distance_{idx} = DTG_{idx2}$

Delete VTCPs

The *Delete VTCPs* function deletes the altitude, speed, and Mach-to-CAS TCPs. The remaining TCPs will only consist of input waypoints, turn-entry, and turn-exit TCPS. This function also removes any flags that associate any remaining TCPs with a speed or altitude change, e.g., a waypoint marked as the 10,000 ft, 250 kt restriction.

Check Turn Validity

The *Check Turn Validity* function is performed after the turn data have been updated and the VTCPs have been deleted. Only input, turn-entry, and turn-exit TCPs should be in the list at this time. The function simple checks that there are no turns within turns by examining the DTG values. If the input flag, *LinearDecelerationFlag*, is true, then a small overlap error of 200 ft is allowed. The variable *LinearDecelerationFlag* is used to denote that speed calculations now include linear deceleration rate estimations.

if (LinearDecelerationFlag = true) offset = 200 / NmiToFeet

else offset = 0

for (i = index number of the first TCP; i < index number of the last TCP; i = i + 1)

if ((DTG$_i$ + offset) < DTG$_{i+1}$) mark this as a fatal error condition

Restore the Crossing Angles

The *Restore the Crossing Angles* function simply replaces the current value for each waypoint's crossing angle with the value that was saved in the function *Save Selected Input Data*.

Calculate Linear Deceleration Rates

This routine is used in the linear deceleration rate calculations to identify the current linear deceleration waypoint and then apply the appropriate change to that waypoint's deceleration rate. Inputs into this routine include the variable *LDRPending*, denoting the initial calculation for each linear deceleration segment.

Find the next linear deceleration rate waypoint whose speed has not been calculated using the secondary function *FindNextLDRWaypoint*, described in a subsequent section.

idx = FindNextLDRWaypoint()

Determine if there is an error at this linear deceleration waypoint.

LdrRateError = false

OtherError = false

if (idx > -1)

The secondary function *LDRRateCheck* will return a value of true for the variable *LdrRateError* if the current linear deceleration segment is not valid and will return a value of true for the variable *OtherError* if a valid speed could not be calculated for any segment.

LDRRateCheck(idx, LDRPending, LdrRateError, OtherError)

If an incomplete or invalid calculation for a linear deceleration has been found and there are no other speed calculation errors, calculate a new deceleration value.

if ((idx > 0) and (OtherError = false)) then

If in the previous calculation the linear deceleration speed adjustment is too small, try a larger deceleration value.

if (LdrRateError) LDRAddRate(idx)

Otherwise, try a smaller deceleration value.

else LDRReduceRate(idx)

Recover the Initial Mach Segments

This function, *Recover the Initial Mach Segments*, attempts to recover the Mach portion of the trajectory if the initial segments should be Mach but have been internally converted to CAS in the function *Meet Cruise CAS Restriction*. This function uses the Mach value that was saved at the start of this program from the first waypoint of the original route. This saved Mach value, *First TCP Mach*, is compared to the Mach equivalent value of the CAS at the initial waypoints and if these Mach values are the same, these waypoints are marked as Mach segments instead of CAS segments.

Only perform this function if the calculated trajectory does not start with a Mach segment but the original route does start with a Mach value.

if ((Mach Segment$_{first\ TCP}$ = false) and (First TCP Mach ≠ 0)) then

if (Crossing CAS$_{first\ TCP}$ = 0) Mach = Crossing Mach$_{first\ TCP}$

else Mach = CasToMach(Crossing CAS$_{first\ TCP}$, Altitude$_{first\ TCP}$)

Determine if this value is close to the original Mach or if there is a different but valid cruise Mach.

DoTest = false

if (Mach ≈ First TCP Mach) DoTest = true

else if ((Mach >= 0.80 Mach) and (Altitude$_{first\ TCP}$ >= 29000 ft)) then

Find the TOD, the speed needs to be the same as the starting speed.

fini = false

i = index number of the first TCP + 1

while ((i < (index number of the last TCP - 1)) and (fini = false))

DoTest = true

if (Altitude$_i$ ≠ Altitude$_{first\ TCP}$) fini = true

else if (CAS$_i$ ≠ CAS$_{first\ TCP}$) then

fini = true

DoTest = false

i = i + 1

end of else if ((Mach >= 0.80 Mach)...

if (DoTest) then

fini = false

i = index number of the first TCP

First Cas = Crossing $CAS_{first\ TCP}$

If there is no Mach transition altitude set, set the transition values.

if (Mach Transition Altitude = 0) then

Mach Descent Mach = First TCP Mach

Mach Transition Cas = First Cas

Mach Transition Altitude = $Altitude_{first\ TCP}$

while ((i < (index number of the last TCP - 1)) and (fini = false))

Test that the CAS computed for the waypoint is the same as the *First Cas*, that except for the first TCP that there is not speed crossing condition at the waypoint, and that the altitude computed for the waypoint is the same as the altitude for the first TCP.

if ((Cas_i *= First Cas) and ((i = index number of the last TCP) or*
((Crossing $Mach_i$ *= 0) and (Crossing* CAS_i *= 0))) and*
($Altitude_i$ *= Crossing* $Altitude_{first\ TCP}$*)) then*

If the previous conditions are true, set this waypoint as a Mach segment.

$Mach\ Segment_i$ *= true*

Change the speed crossing values for the first TCP.

if (Crossing CAS_i *> 0) then*

Crossing CAS_i *= 0*

Crossing $Mach_i$ *= First TCP Mach*

end of if ((Cas_i *= First Cas)...)*

else

fini = true

if ((i > index number of the first TCP) and ($VSegType_{i-1}$ = SPEED))

$VSegType_{i-1}$ = MACH CAS

i = i + 1

end of while ((i < (index number of the last TCP - 1)) and (fini = false))

end of if (DoTest)

end of if ((Mach $Segment_{first\ TCP}$ = false) and (First TCP Mach ≠ 0))

else if ((Mach $Segment_{last\ TCP}$ = true) and (Mach Transition Mach == 0) and (Mach $Cross_{last\ TCP}$ > 0))

There are only Mach segments, make sure the transition Mach is valid.

Mach Transition Mach = Mach $Cross_{last\ TCP}$

Insert CAS Descent VTCPs

This function inserts vertical TCPs between constant CAS descent waypoints to improve the TAS estimation when using the data provided by this algorithm. This updating occurs at 3,000 ft intervals.

Update Altitude = 3000

Find the first CAS point.

j = 0

while ((Mach $Segment_i$ = true) and ($VSegType_j$ ≠ MACH CAS) and (j < index number of the last TCP)) j = j + 1

for (i = j; i < (index number of the last TCP - 1); i = i + 1)

$DeltaZ = Altitude_i - Altitude_{i+1}$

Update at 3000 ft intervals but skip the update if the waypoint is within 500 ft of the test altitude.

if ((DeltaZ ≥ (Update Altitude + 500)) and ($Cas_i \approx Cas_{i+1}$)) then

$z = Altitude_i$ - Update Altitude

$dx = DTG_i - DTG_{i+1}$

*a = arctangent2 (DeltaZ, NmiToFeet * dx)*

$d = DTG_i$ - Update Altitude / tan(a) / NmiToFeet

Compute the ground track at distance *d* along the trajectory and save it as *Saved Ground Track*.

Saved Ground Track = GetTrajGndTrk(d)

k = i + 1

Insert a new VTCP at location k in the TCP list. The VTCP is inserted between TCP_{k-1} and TCP_k from the original list. The function *InsertTcp* should be appropriate for the actual data structure implementation of this function.

InsertTcp(k)

Update the TCP-type data in the new TCP.

$TcpType_k$ = *VTCP*

$VSegType_k$ = *TAS ADJUSTMENT*

$TurnType_k$ = *NO TURN*

Update the crossing data in the new waypoint.

Crossing $Mach_k = 0$

Crossing $CAS_k = 0$

Crossing $Rate_k = 0$

$CAS_k = CAS_{k+1}$

$DTG_k = d$

$Altitude_k = z$

$Mach_k = CasToMach(CAS_k, Altitude_k)$

Mach Segment$_k$ = *false*

Crossing $Angle_k$ = *Crossing* $Angle_{k+1}$

Ground $Track_k$ = *Saved Ground Track*

Compute and add the wind data at this waypoint.

$GenerateWptWindProfile(DTG_k, TCP_k)$

Compute the wind at the waypoint altitude and then waypoint's ground speed.

$InterpolateWindWptAltitude(Wind\ Profile_k, Altitude_k, Ws, Wd, Td)$

$Ground\ Speed_k = ComputeGndSpeedUsingTrack(CAS_k, Ground\ Track_{k-1}, Altitude_k, Ws, Wd, Td)$

Compute TCP Times

The function *Compute TCP Times* calculates the time to each TCP. The calculations begin at the runway (the last TCP), working backwards, and compute the TTG to each TCP.

$TTG_{last\ TCP} = 0$

for (i = index number of the last TCP; i > index number of the first TCP; i = i - 1)

Average Ground Speed = $(Ground\ Speed_{i-1} + Ground\ Speed_i) / 2$

$x = DTG_{i-1} - DTG_i$

Test for an error condition where the distance is less than 0. This error only occurs if the segment ends overlap.

if (x < 0) then

Find the previous input waypoint in case it is needed in a later test. Also determine if this previous waypoint is an RF turn point.

PreviousIsRf = false

fini = false

j = i - 1

while (fini = false)

if (j < index number of the first TCP) fini = true

else if (($TcpType_j$ *= INPUT) and (*$TurnType_j$ *= RF TURN CENTER)) then*

PreviousIsRf = true

fini = true

else if ($TcpType_j$ *= INPUT) fini = true*

j = j - 1

end of while (fini = false)

If the distance is close to 0, e.g., within 500 ft for a normal segment pair, set the distance to the previous distance value and ignore the error.

if (x ≥ (-500 ft / NmiToFeet)) then

$DTG_i = DTG_{i-1}$

x = 0

Allow a larger margin of error of 1500 ft for the beginning of an RF turn.

else if ((x ≥ -1500 ft / NmiToFeet) and ($TurnType_i$ = TURN START) and ($Center\ Of\ Turn\ Latitude_i \neq 0$)) then

$DTG_i = DTG_{i-1}$

$x = 0$

Allow a larger margin of error of 1500 ft if the end of the previous segment is the end of an RF turn and it overlaps the start of another turn.

else if ((x ≥ -1500 ft / NmiToFeet) and ($TurnType_i$ = TURN START) and (i > index number of the first TCP) and ($TurnType_{i-1}$ = turn-exit) and PreviousIsRf) then

Overwrite the previous end of turn data with the subsequent start of turn data.

$DTG_{i-1} = DTG_i$

$Altitude_{i-1} = Altitude_i$

$CAS_{i-1} = CAS_i$

$Ground\ Speed_{i-1} = Ground\ Speed_i$

$Ground\ Track_{i-1} = Ground\ Track_i$

$Mach_{i-1} = Mach_i$

$Mach\ Segment_{i-1} = Mach\ Segment_i$

$x = 0$

else mark this as a fatal error condition

end of if (x < 0)

*Delta Time = (3600 sec/hr) * x / Average Ground Speed*

$TTG_{i-1} = TTG_i + Delta\ Time$

Compute TCP Latitude and Longitude Data

With the exception of the input waypoints, the *Compute TCP Latitude and Longitude Data* function computes the latitude and longitude data for all of the TCPs.

In Turn = false

Last Base = index number of the first TCP

Next Input = index number of the first TCP

Turn Index = index number of the first TCP

Turn is Clockwise = true

Turn Adjustment = 0

Base Latitude = $Latitude_{Last\ Base}$

Base Longitude = $Longitude_{Last\ Base}$

for (i = index number of the first TCP; i ≤ index number of the last TCP; i = i + 1)

if ($TcpType_i$ = BEGIN TURN) then

Turn Adjustment = 0

InTurn = true

Find the major waypoint for this turn.

Next Input = i + 1

while (($TcpType_{Next\ Input}$ ≠ INPUT) and (Next Input ≤ index number of the last TCP))
Next Input = Next Input + 1

Turn Index = Next Input

a = DeltaAngle($Ground\ Track_i$, $Ground\ Track_{Next\ Input}$)

x = Turn Data $Turn\ Radius_{Turn\ Index}$ / cosine(a)

if (a > 0°) Turn Clockwise =true

else Turn Clockwise = false

if (Turn Clockwise) a1 = $Ground\ Track_{Turn\ Index}$ + 90°

else a1 = $Ground\ Track_{Turn\ Index}$ - 90°

Now compute the relative latitude and longitude values. The function *RelativeLatLon* is described in a subsequent section.

RelativeLatLong($Latitude_{Turn\ Index}$, $Longitude_{Turn\ Index}$, a1, x), returning Center Latitude and Center Longitude

end of if ($TcpType_i$ = BEGIN TURN)

if (In Turn) then

Turn Adjustment = 0

if (Turn Clockwise) a1 = Ground Track$_i$ - 90°

else a1 = Ground Track$_i$ + 90°

if (TcpType$_i$ = INPUT) then

Turn Data Center Latitudei = Center Latitude

Turn Data Center Longitudei = Center Longitude

RelativeLatLong(Center Latitude, Center Longitude, a1, Turn Data Turn Radius$_{Turn\ Index}$), returning Turn Data Latitude$_i$ and Turn Data Longitude$_i$

end of if (TcpType$_i$ = INPUT)

else RelativeLatLon(Center Latitude, Center Longitude, a1, Turn Data Turn Radius$_{Next\ Input}$), returning Latitude$_i$ and Longitude$_i$

if (TcpType$_i$ = END TURN) then

Turn Adjustment = Turn Data Straight Distance2$_{Turn\ Index}$ - Turn Data Path Distance2$_{Turn\ Index}$

In Turn = false

Last Base = Next Input

Base Latitude = Latitude$_{Last\ Base}$

Base Longitude = Longitude$_{Last\ Base}$

end of if (In Turn)

else

if (TcpType$_i$ = INPUT) then

Turn Adjustment = 0

Last Base = i

Base Latitude = Latitude$_{Last\ Base}$

Base Longitude = Longitude$_{Last\ Base}$

else

RelativeLatLong(Base Latitude, Base Longitude, Ground Track$_{i-1}$, DTG$_{Last\ Base}$ - DTG$_i$ + Turn Adjustment), returning Latitude$_i$ and Longitude$_i$

end of for (i = index number of the first TCP; i ≤ index number of the last TCP; i = i + 1)

Description of Secondary Functions

The secondary functions are listed in alphabetical order. Note that standard aeronautical functions, such as CAS to Mach conversions, *CasToMach*, are not expanded in this document but may be found numerous references, e.g., reference 35. It may also be of interest to include atmospheric temperature or temperature deviation in the wind data input and calculate the temperature at the TCP crossing altitudes to improve the calculation of the various speed terms.

AdjustAngle

The function *AdjustAngle* adjusts the angle a such that $0° \geq a \geq 360°$.

$x = a / 360$

i = *truncate x to an integer*

$x = x - i$

$x = 360 * x$

if $(x \leq 0)$ $x = 360 + x$

$a = x$

BodDecelerationDistance

The function *BodDecelerationDistance* estimates the distance required for the special case of a deceleration to a CAS restricted waypoint from the Mach-to-CAS transition. This function is invoked from *ComputeDescentAccelDecel*, which passes in the index number for the bottom-of-descent (TOD) waypoint, *BodIndex*, the Mach transition to CAS altitude, *Mach Transition Altitude*, and the CAS at the Mach transition to CAS, *TransitionCas*. The function returns the distance from the index point of the deceleration, *Distance*.

Estimate the distance to the new Mach value. Begin by finding the time to do the deceleration.

$t = (TransitionCas - Crossing\ CAS_{BodIdx}) / Crossing\ Rate_{BodIdx}$

Compute the wind speed and direction at the current altitude.

InterpolateWindWptAltitude(Wind Profile $_{BodIdx}$*, Altitude* $_{BodIdx}$*, Ws, Wd, Td)*

Calculate the ground track at the current point.

if (WptInTurn(BodIdx)) track = $Ground\ Track_{BodIdx-1}$

else track = $Ground\ Track_{BodIdx}$

Calculate the ground speed over this segment.

BodGs = ComputeGndSpeedUsingTrack($Crossing\ CAS_{BodIdx}$*, track,* $Altitude_{BodIdx}$*, Ws, Wd, Td)*

DescentGs = ComputeGndSpeedUsingTrack(TransitionCas, track, Mach Transition Altitude, Ws, Wd, Td)

Calculate the average groundspeed, *AvgGS*.

AvgGs = (BodGs + DescentGs) / 2

The distance estimate is *AvgGs * t* .

*Distance = AvgGs * t / (3600 sec/hr)*

ComputeTodAcceleration

The *ComputeTodAcceleration* is for the special case where the descent Mach is higher than the cruise Mach> This function generates the waypoint speeds between the TOD TCP and the VTCP where the descent Mach is achieved. Inputs into this function include the TCP index value, *TodIdx*, for the TOD TCP and the TCP index value, *AccelIdx*, for the VTCP that ends the descent acceleration.

Save the relevant speeds and deceleration value.

PriorSpd = $Mach_{TodIdx}$

TestSpd = Crossing $Mach_{AccelIdx}$

TestRate = - CasToMach(Crossing CAS_{TodIdx}, $Altitude_{AccelIdx}$)

k = AccelIdx

while (k > (TodIdx + 1))

Make an estimate of the distance required to meet the speed change.

Calculate the time to do the deceleration.

t = (PriorSpd - TestSpd) / TestRate

Compute the wind speed and direction at the current altitude.

InterpolateWindWptAltitude(Wind $Profile_k$, $Altitude_k$, Ws, Wd, Td)

Save the groundtrack value.

if (WptInTurn(k)) track = Ground $Track_k$

else track = Ground $Track_{k-1}$

CurrentGs = ComputeGndSpeedUsingMachAndTrack(TestSpd, track, $Altitude_k$, Ws, Wd, Td)

Compute the wind speed and direction at the prior altitude.

InterpolateWindWptAltitude(Wind $Profile_{k-1}$, $Altitude_{k-1}$, Ws, Wd, Td)

Compute the groundspeed at the prior point.

$PriorGs = ComputeGndSpeedUsingMachAndTrack(PriorSpd, Ground\ Track_{k-1}, Altitude_{k-1}, Ws, Wd, Td)$

Calculate the average groundspeed

$AvgGs = (PriorGs + CurrentGs) / 2$

The distance estimate (dx) is AvgGs * T.

$dx = AvgGs * t / (3600\ sec/hr)$

Now recompute the distance required to meet the speed using the estimated distance *dx* from the previous calculation. Begin by computing the altitude, *AltD*, at distance *dx*.

$if\ (Altitude_k \geq Altitude_{k-1})\ AltD = Altitude_k$

else

$AltD = dx * tan(Crossing\ Angle_k) + Altitude_k$

$if\ (AltD \geq Altitude_{k-1})\ AltD = Altitude_k$

Calculate the deceleration value.

$MRate1 = - CasToMach(Crossing\ Rate_{Accelldx}, Altitude_k)$

$MRate2 = - CasToMach(Crossing\ Rate_{Accelldx}, AltD)$

$TestRate = (MRate1 + MRate2) / 2$

$t = (PriorSpd - TestSpd) / TestRate$

$if\ (t < 0)\ t = 0$

$dx = AvgGs * t / (3600\ sec/hr)$

Note that the previous time 't' for Cas is still valid.

$x = DTG_k + dx$

Compute the winds at *AltitudeD* and at distance *x*.

GetWindAtAltitudeDistance(AltD, x, Ws2, Wd2, Td2)

Compute the track angle.

track2 = GetTrajGndTrk(x)

Compute the groundspeed at *AltD*.

PriorGs = ComputeGndSpeedUsingMachAndTrack(PriorSpd, track2, AltD, Ws2, Wd2, Td2)

Calculate the average groundspeed, *AvgGS*.

AvgGs = (PriorGs + CurrentGs) / 2

The distance estimate, *dx*, is *AvgGs * t*.

*dx = AvgGs * t / (3600 sec/hr)*

If a waypoint exists prior to this distance, compute and insert its speed and calculate again.

if $(DTG_{k-1} < (DTG_k + dx))$ *then*

Compute the speed at the waypoint using v^2 = v0^2 + 2ax to get v.

Begin by calculating the headwinds at the end point.

$HeadWind2 = Ws2 * cos(Wd2 - Ground\ Track_{k-1})$

$dx = DTG_{k-1} - DTG_k$

$MRate1 = CasToMach(Crossing\ Rate_{AccelIdx}, Altitude_k)$

$MRate2 = CasToMach(Crossing\ Rate_{AccelIdx}, Altitude_{k-1})$

TestRate = -(MRate1 + MRate2) / 2

$Mach_{k-1} = EstimateNextMach(TestSpd, CurrentGS, PriorSpd, HeadWind2, Altitude_k, dx, TestRate, Td)$

if $(Mach_{k-1} < PriorSpd)\ Mach_{k-1} = PriorSpd$

Check to determine if the constraint has been met.

if $((k-1) = TodIdx)\ Mach_{TodIdx} = Crossing\ Mach_{TodIdx}$

$TestSpd = Mach_{k-1}$

$Mach\ Segment_{k-1} = true$

Go to the next TCP.

k = k - 1

end of if $(DTG_{k-1} < (DTG_k + dx))$

ComputeGndSpeedUsingMachAndTrack

The *ComputeGndSpeedUsingMachAndTrack* function computes a ground speed from track angle (versus heading), *track*, Mach, *Mach*, altitude, *Altitude*, and wind data, *Wind Speed*, *Wind Direction, and Temperature Deviation*.

CAS = MachToCas(Mach, Altitude)

Ground Speed = ComputeGndSpeedUsingTrack(CAS, track, Altitude, Wind Speed, Wind Direction, Temperature Deviation)

return Ground Speed

ComputeGndSpeedUsingTrack

The *ComputeGndSpeedUsingTrack* function computes a ground speed from track angle (versus heading), *track*, CAS, *CAS*, altitude, *Altitude*, and wind data, *Wind Speed*, *Wind Direction, and Temperature*.

b = DeltaAngle(track, Wind Direction)

if (CAS ≤ 0) r = 0

*else r = (Wind Speed / CasToTas Conversion(CAS, Altitude, Temperature)) * sine(b),*

where *CasToTas Conversion* is a standard conversion routine from CAS to TAS. If no temperature value is utilized in the input, then a standard temperature is assumed.

Limit the correction to something reasonable.

*if (|r| > 0.8) r = 0.8 * r / |r|*

heading = track + arcsine(r)

a = DeltaAngle(heading, Wind Direction)

TAS = CasToTas Conversion(CAS, Altitude, Temperature)

$$Ground\ Speed = (Wind\ Speed^2 + TAS^2 - 2 * Wind\ Speed * TAS * cosine(a))^{0.5}$$

ComputeGndTrk

The *ComputeGndTrk* function computes the ground track at the along-path distance equal to *distance*, where distance must lie between TCP_{i-1} and TCP_{i+1}. It is assumed that the value for $Ground\ Track_i$ is invalid. The function uses a linear interpolation based on DTG_{i-1} and DTG_{i+1}, with the index value *i* input into the function and where the distance, *distance,* must lie between these points.

$d = DTG_{i-1} - DTG_{i+1}$

$if\ (d \leq 0)\ Ground\ Track = Ground\ Track_{i-1}$

else

$a = (1 - (distance - DTG_{i+1}) / d) * DeltaAngle(Ground\ Track_{i-1}, Ground\ Track_{i+1})$

$Ground\ Track = Ground\ Track_{i-1} + a$

ComputeTcpCas

The index variable *cc* is passed into and out of the *ComputeTcpCas* function. Beginning with the last waypoint, this function computes the CAS at each previous TCP and inserts any additional speed TCPs that may be required to denote a change in the speed profile. The function uses the current speed constraint, searches backward for the previous constraint, and then computes the distance required to meet this previous constraint. The speeds for all of the TCPs within this distance are computed and added to the data for the TCPs. If the along-path distance to meet the previous constraint is not at a TCP, a new speed VTCP is inserted at this distance. Because there is no general closed form solution to compute distances to meet the deceleration constraints, an iterative technique is used in this function. This function is performed in the following steps:

While ((cc > index number of the first TCP) and ($VSegType_{cc} \neq$ MACH CAS))

Determine if the previous constraint cannot be met.

If (CAS_{cc} > Crossing CAS_{cc}) then

If this is the last pass through the algorithm, mark this as a fatal error condition

CAS_{cc} = Crossing CAS_{cc}

Find the prior waypoint index number *pc* that has a CAS constraint, e.g., a crossing CAS (*Crossing* $CAS_{pc} \neq 0$). This may not always be the previous (i.e., *cc - 1*) waypoint.

The initial condition is the previous TCP.

pc = cc - 1

fini - false

while ((fini = false) and (pc > index number of the first TCP))

if (($VSegType_{pc}$ = MACH CAS) or (Crossing CAS_{pc} = 0) or
((pc > (index number of the last TCP - 1)) and
($VSegType_{pc+1}$ = TOD DECELERATION))) fini = true

else pc = pc - 1

Save the previous crossing speed. Determine if there is no prior CAS crossing constraint because it is a Mach segment.

if ((Crossing CAS_{pc} = 0) and (Crossing $Mach_{pc}$ > 0))

PriorSpd = MachToCas(Crossing $Mach_{pc}$, $Altitude_{pc}$)

else Prior Speed = Crossing CAS_{pc}

Save the current crossing speed (*Test Speed*) at TCP_{cc} and the deceleration rate (*Test Rate*) noting that the first and last waypoints always have speed constraints and except for the first TCP, all constrained speed points must have deceleration rates.

Test Speed = Crossing CAS_{cc}

Test Rate = Crossing Rate$_{cc}$

Compute all of the TCP speeds between the current TCP and the previous crossing waypoint.

k = cc

while (k > pc)

If the previous speed has already been reached, set the remaining TCP speeds to the previous speed.

if (Prior Speed ≤ Test Speed) then

for (k = k - 1; k > pc; k = k - 1)

CAS$_k$ = Test Speed

Mach$_k$ = CasToMach(CAS$_k$, Altitude$_k$)

Set the speeds at the last test point.

CAS$_{pc}$ = Test Speed

if (Mach$_{pc}$ = 0) Mach$_{pc}$ = CasToMach(CAS$_{pc}$, Altitude$_{pc}$)

else

Estimate the distance required to meet the crossing restriction using the winds at the current altitude. This is a first-estimation.

Compute the time to do the deceleration.

t = (Prior Speed - Test Speed) / Test Rate

Compute the wind speed and direction at the current altitude.

InterpolateWindWptAltitude(Wind Profile$_k$, Altitude$_k$,Wind Speed1, Wind Direction1, Temperature Deviation1)

The ground track at the current point is,

if (WptInTurn(k)) Track = Ground Track$_k$

else Track = Ground Track$_{k-1}$

Current Ground Speed = ComputeGndSpeedUsingTrack(Test Speed, Track, Altitude$_k$,Wind Speed1, Wind Direction1, Temperature Deviation1)

Compute the wind speed and direction at the prior altitude.

InterpolateWindWptAltitude(Wind Profile$_{k-1}$, Altitude$_k$,Wind Speed1, Wind Direction1, Temperature Deviation1)

The ground speed at the prior point.

Prior Ground Speed = ComputeGndSpeedUsingTrack(Prior Speed, GndTrack$_{k-1}$, Altitude$_{k-1}$, Wind Speed1, Wind Direction1, Temperature Deviation1)

Average Ground Speed = (Prior Ground Speed + Current Ground Speed) / 2

The distance estimate, *dx*, is *Average Ground Speed * t*.

*dx = Average Ground Speed * t / (3600 sec/hr)*

Recalculate the distance required to meet the speed using the previous estimate distance *dx.*

Begin by computing the altitude*, AltD,* at distance *dx.*

if (Altitude$_k$ ≥ Altitude$_{k-1}$) AltD = Altitude$_k$

else

*AltD = (NmiToFeet * dx) * tangent(Crossing Angle$_k$) + Altitude$_k$*

if (AltD ≥ Altitude$_{k-1}$) AltD = Altitude$_k$

The new distance *x* is *DTG$_k$ + dx*.

x = DTG$_k$ + dx

Compute the winds at *AltD* and distance *x*.

GetWindAtAltitudeDistance(AltD, x, Wind Speed2, Wind Direction2, Temperature Deviation2)

The track angle at this point, with *GetTrajGndTrk* defined in this section:

Track2 = GetTrajGndTrk(x)

The ground speed at altitude *AltD* is then,

Prior Ground Speed = ComputeGndSpeedUsingTrack(Prior Speed, Track2, AltD, Wind Speed2, Wind Direction2, Temperature Deviation2)

Average Ground Speed = (Prior Ground Speed + Current Ground Speed) / 2

*dx = Average Ground Speed * t / (3600 sec/hr)*

If there is a TCP prior to *dx*, compute and insert its speed.

If the distance is very close to the waypoint, just set the speed.

if $((DTG_{k-1} < (DTG_k + dx))$ *then*

Compute the speed at the waypoint using $v^2 = v_0^2 + 2ax$ to get v.

The headwind at the end point is,

*HeadWind2 = Wind Speed2 * cosine(Wind Direction2 - Ground* $Track_{k-1}$*)*

$dx = DTG_{k-1} - DTG_k$

The value of CAS_{k-1} is computed using function *EstimateNextCas*, described in this section.

CAS_{k-1} *= EstimateNextCas(Test Speed, Current Ground Speed, false, Prior Speed, Head Wind2,* $Altitude_k$*, dx,* $Crossing\ Rate_{cc,}$ *Temperature Deviation1)*

Determine if the constraint is met.

if ((k-1) = pc) then

Determine the allowable crossing window, accounting for special conditions.

if (((pc + 1) < index number of the last TCP) and $(VSegType_{pc}$ *= MACH CAS))*

CrossingWindow = 5

else CrossingWindow = 2

If this is the altitude CAS restriction and the calculated speed is below the required crossing speed, then ignore this error.

if (($VSegType_{pc}$ *== ALTITUDE CAS RESTRICTION) and* $(Cas_{pc} < Crossing\ CAS_{pc}))$

ignore = true

else ignore = false

Was the crossing window speed met? If not, set this as an error.

if ((ignore = false) and ($|CAS_{pc} - Crossing\ CAS_{pc}| > CrossingWindow))$

mark this as a fatal error condition

Always set the crossing exactly to the crossing speed.

$CAS_{pc} = Crossing\ CAS_{pc}$

Set the test speed to the computed speed.

Test Speed = CAS_{k-1}

Back up the index counter to the next intermediate TCP.

$k = k - 1$

end of if (($DTG_{k-1} < (DTG_k + dx)$)

else

The constraint occurs between this TCP and the previous TCP. A new VTCP needs to be added at this point.

The along path distance *d* where the VTCP is to be inserted is:

$d = DTG_k + dx$

Save the ground track value at this distance.

Saved Ground Track = GetTrajGndTrk(d)

Insert a new VTCP at location *k* in the TCP list. The VTCP is inserted between TCP_{k-1} and TCP_k from the original list. The function *InsertTcp* should be appropriate for the actual data structure implementation of this function.

InsertTcp(k)

Update the data for the new VTCP which is now TCP_k.

$TcpType_k$ = *VTCP*

if ($VSegType_k$ = *NO TYPE*) $VSegType_k$ = *SPEED*

$TurnType_k$ = *NO TURN*

$DTG_k = d$

The altitude at this point is computed as follows, recalling that the new waypoint is TCP_k:

if ($Altitude_{k+1} \geq Altitude_{k-1}$) $Altitude_k = Altitude_{k-1}$

else $Altitude_k$ = (*NmiToFeet* * *dx*) * *tangent*($Crossing\ Angle_{k+1}$) + $Altitude_{k+1}$

CAS_k = *Prior Speed*

Add the ground track data which must be computed if the new VTCP occurs within a turn. The functions *WptInTurn* and *ComputeGndTrk* are described in subsequent sections.

if (*WptInTurn*(k)) $Ground\ Track_k$ = *ComputeGndTrk*(k, d)

else Ground Track$_k$ = Saved Ground Track

Compute and add the wind data at distance *d* along the path to the data of *TCP$_k$*.

GenerateWptWindProfile(d, TCP$_k$)

Test Speed = Prior Speed

Since *TCP$_k$*, has now been added prior to *pc*, the current constraint counter *cc* needs to be incremented by 1 to maintain its correct position in the list.

cc = cc + 1

end of while k > pc.

Now go to the next altitude change segment on the profile.

cc = k

end of while cc > index number of the first TCP

ComputeTcpMach

The index variable cc is passed into and out of the *ComputeTcpMach* function. This function is similar to *ComputeTcpCas* with the exception that the computed Mach rate will need to be recomputed with any change of altitude. Beginning with the last Mach waypoint (the Mach waypoint that is closest to the runway in terms of DTG), this function computes the Mach at each previous TCP and inserts any additional speed TCPs that may be required to denote a change in the speed profile. The function uses the current speed constraint, searches backward for the previous constraint, and then computes the distance required to meet this previous constraint. The speeds for all of the TCPs within this distance are computed and added to the data for the TCPs. If the along-path distance to meet the previous constraint is not at a TCP, a new speed VTCP is inserted at this distance. Because there is no general closed form solution to compute distances to meet the deceleration constraints, an iterative technique is used in this function. This function is performed in the following steps:

Initialize descent acceleration flag.

HadDescentAccelSegment = false

while (cc > index number of the first TCP)

Determine if the previous constraint cannot be met.

if (Mach$_{cc}$ > Crossing Mach$_{cc}$) then

Ignore some errors regarding the TOD speed.

if ((HadDescentAccelSegment = false) or (VSegType$_{cc}$ ≠ TOD ALTITUDE)) then

mark this as a fatal error condition

Insert a reasonable value for the Mach so that the calculations can continue.

$Mach_{cc}$ = *Crossing* $Mach_{cc}$

end of if ($Mach_{cc}$ > *Crossing* $Mach_{cc}$*)*

Find the prior waypoint index number *pc* that has a Mach constraint, e.g., a crossing Mach (*Crossing* $Mach_{pc} \neq 0$). This may not always be the previous (i.e., *cc - 1*) waypoint.

Initial condition is the previous TCP.

pc = cc - 1

finished = false

if ($VSegType_{pc}$ *= TOD ACCELERATION) Accelerating = true*

else Accelerating = false

while ((fini = false) and (pc > index number of the first TCP))

if ($VSegType_{pc}$ *= MACH CAS) fini = true*

if ((pc > (index number of the last TCP - 1)) and
($VSegType_{pc+1}$ *= TOD DECELERATION) fini = true*

if ((Accelerating - true) and ($VSegType_{pc}$ *= TOD ALTITUDE)) fini = true*

if (Crossing $Rate_{pc}$ *> 0) fini = true*

if (fini = false) pc = pc - 1

end of while ((fini = false) and (pc > index number of the first TCP))

if (Accelerating = true)

ComputeTodAcceleration(pc, cc)

HadDescentAccelSegment = true

k = pc

If not accelerating, just perform the normal routine.

else

Save the previous crossing speed,

Prior Speed = Crossing $Mach_{pc}$

Handle the special case of a deceleration at the Mach /CAS transition.

if ((AllowTodDeceleration = true) and ($VSegType_{cc}$ = MACH CAS) and
($Crossing\ Mach_{cc}$ = 0)) then

fini = false

k = cc - 1

Test Speed = $Mach_k$

Find the last Mach value.

while ((fini = false) and (k ≥ index number of the first TCP))

if ($Crossing\ Mach_k$ > 0) then

fini = true

TestSpd = $Mach_k$

k = k - 1

$Crossing\ Mach_{cc}$ = TestSpd

Set the deceleration value to a default rate of 0.25 kt/sec for a TOD deceleration to the descent Mach.

TestRate = 0.25 kt/sec

end of if ((AllowTodDeceleration = true) and ($VSegType_{cc}$ = MACH CAS) and
($Crossing\ Mach_{cc}$ = 0)) then

else

if (($VSegType_{cc}$ = TOD ALTITUDE) and ($Crossing\ Machs_{cc}$ = 0)) TestSpd = $Mach_{cc}$

else TestSpd = $Crossing\ Mach_{cc}$

Convert the rate to a Mach value.

TestRate = CasToMach($Crossing\ Rate_{cc}$, $Altitude_{cc}$)

k = cc

Compute all of the TCP speeds between the current TCP and the previous crossing waypoint.

k = cc

while (k > pc)

If the previous speed has already been reached, set the remaining TCP speeds to the previous speed.

if (Prior Speed ≤ Test Speed) then

for (k = k - 1; k > pc; k = k - 1)

$Mach_k$ = *Test Speed*

CAS_k = *MachToCas(*$Mach_k$, $Altitude_k$*)*

$Mach\ Segment_k$ = *true*

Set the speeds at the last test point.

$Mach_{pc}$ = *Test Speed*

CAS_{pc} = *MachToCas(*$Mach_{pc}$, $Altitude_{pc}$*)*

end of if (Prior Speed ≤ Test Speed) then

else

Estimate the distance required to meet the crossing restriction using the winds at the current altitude. This is a first-estimation.

Compute the time to do the deceleration.

t = (Prior Speed - Test Speed) / Test Rate

Compute the wind speed and direction at the current altitude.

InterpolateWindWptAltitude(Wind Profile$_k$*, Altitude*$_k$*,Wind Speed1, Wind Direction1, Temperature Deviation1)*

The ground track at the current point is,

if (WptInTurn(k)) Track = Ground Track$_k$

else Track = Ground Track$_{k-1}$

Current Ground Speed = ComputeGndSpeedUsingMachAndTrack(Test Speed, Track, Altitude$_k$*, Wind Speed1, Wind Direction1, Temperature Deviation1)*

Compute the wind speed and direction at the prior altitude.

InterpolateWindWptAltitude(Wind Profile$_{k-1}$*, Altitude*$_k$*,Wind Speed1, Wind Direction1, Temperature Deviation1)*

The ground speed at the prior altitude and speed is,

Prior Ground Speed = ComputeGndSpeedUsingMachAndTrack(Prior Speed, GndTrack$_{k-1}$*, Altitude*$_{k-1}$*, Wind Speed1, Wind Direction1, Temperature Deviation1)*

Average Ground Speed = (Prior Ground Speed + Current Ground Speed) / 2

The distance estimate, *dx*, is *Average Ground Speed * t*.

*dx = Average Ground Speed * t / (3600 sec/hr)*

Compute the distance required to meet the speed using the previous estimate distance *dx*.

Begin by computing the altitude*, AltD,* at distance *dx*.

if ($Altitude_k \geq Altitude_{k-1}$) $AltD = Altitude_k$

else

$AltD = (NmiToFeet * dx) * tangent(Crossing\ Angle_k) + Altitude_k$

Compute the average Mach rate.

$MRate1 = CasToMach(Crossing\ Rate_{cc}, Altitude_k)$

$MRate2 = CasToMach(Crossing\ Rate_{cc}, AltD)$

Test Rate = (MRate1 + MRate2) / 2

t = (Prior Speed - Test Speed) / Test Rate

The new distance *x* is $DTG_k + dx$.

$x = DTG_k + dx$

Compute the winds at *AltD* and distance *x*.

GetWindAtAltitudeDistance(AltD, x, Wind Speed2, Wind Direction2, Temperature Deviation2)

The track angle at this point, with *GetTrajGndTrk* defined in this section, is:

Track2 = GetTrajGndTrk(x)

The ground speed at altitude *AltD* is then,

Prior Ground Speed = ComputeGndSpeedUsingMachAndTrack(Prior Speed, Track2, AltD, Wind Speed2, Wind Direction2, Temperature Deviation2)

Average Ground Speed = (Prior Ground Speed + Current Ground Speed) / 2

*dx = Average Ground Speed * t / (3600 sec/hr)*

If there is a TCP prior to *dx*, compute and insert its speed.

If the distance is very close to the waypoint, just set the speed.

if $((DTG_{k-1} < (DTG_k + dx))$ *then*

Compute the speed at the waypoint using $v^2 = v_0^2 + 2ax$ to get v.

The headwind at the end point is,

*HeadWind2 = Wind Speed2 * cosine(Wind Direction2 - Ground Track*$_{k-1}$*)*

$dx = DTG_{k-1} - DTG_k$

Compute the average Mach rate.

MRate1 = CasToMach(Crossing Rate$_{cc}$*, Altitude*$_k$*)*

MRate2 = CasToMach(Crossing Rate$_{cc}$*, Altitude*$_{k-1}$*)*

Test Rate = (MRate1 + MRate2) / 2

The value of $Mach_{k-1}$ is computed using function *EstimateNextMach*, described in this section.

$Mach_{k-1}$ *= EstimateNextMach(Test Speed, Current Ground Speed, Prior Speed, Head Wind2, Altitude*$_k$*, dx, Test Rate)*

Determine if the constraint is met.

if ((k-1) = pc) then

Was the crossing speed met within 0.002 Mach? If not, set this as an error.

if (|Mach$_{pc}$ *- Crossing Mach*$_{pc}$*| > 0.002) mark this as a fatal error condition*

Always set the crossing exactly to the crossing speed.

$Mach_{pc}$ *= Crossing Mach*$_{pc}$

Set the test speed to the computed speed.

Test Speed = $Mach_{k-1}$

Mach Segment$_{k-1}$ *= true*

Back up the index counter to the previous intermediate TCP.

k = k - 1

end of if $((DTG_{k-1} < (DTG_k + dx))$

else

The constraint occurs between this TCP and the previous TCP. A new VTCP needs to be added at this point.

The along path distance *d* where the VTCP is to be inserted is:

$d = DTG_k + dx$

Save the ground track value at this distance.

Saved Ground Track = GetTrajGndTrk(d)

Insert a new VTCP at location *k* in the TCP list. The VTCP is inserted between TCP_{k-1} and TCP_k from the original list. The function *InsertTcp* should be appropriate for the actual data structure implementation of this function.

InsertTcp(k)

Update the data for the new VTCP which is now TCP_k.

$TcpType_k = VTCP$

if $(VSegType_k = NO\ TYPE)$ $VSegType_k = SPEED$

$TurnType_k = NO\ TURN$

$DTG_k = d$

The altitude at this point is computed as follows, recalling that the new waypoint is TCP_k:

if $(Altitude_{k+1} \geq Altitude_{k-1})$ $Altitude_k = Altitude_{k-1}$

else $Altitude_k = (NmiToFeet * dx) * tangent(Crossing\ Angle_{k+1}) + Altitude_{k+1}$

$Mach_k = Prior\ Speed$

$Mach\ Segment_k = true$

Add the ground track data which must be computed if the new VTCP occurs within a turn. The functions *WptInTurn* and *ComputeGndTrk* are described in subsequent sections.

if $(WptInTurn(k))$ $Ground\ Track_k = ComputeGndTrk(k, d)$

else $Ground\ Track_k = Saved\ Ground\ Track$

Compute and add the wind data at distance *d* along the path to the data of TCP_k.

GenerateWptWindProfile(d, TCP_k*)*

Test Speed = Prior Speed

Since TCP_k, has now been added prior to *pc*, the current constraint counter *cc* needs to be incremented by 1 to maintain its correct position in the list.

cc = cc + 1

end of while (k > pc)

end of else (not accelerating)

Now go to the next altitude change segment on the profile.

cc = k

end of while cc > index number of the first TCP.

Make sure that the waypoints get marked correctly if there are no CAS waypoints.

if ((begin > index number of the first TCP) and (cc = index number of the first TCP)) then

for (k = index number of the first TCP; k < begin; k++)

$Mach\ Segment_k = true$

DeltaAngle

The *DeltaAngle* function returns angle *a*, the difference between *Angle1* and *Angle2*. The returned value may be negative, i.e., -180 degrees ≥ *DeltaAngle* ≥ 180 degrees.

a = Angle2 - Angle1

Adjust *a* such that 0° ≥ *a* ≥ 360°.

AdjustAngle(a)

if (a > 180°) a = a - 360°

return a

DoTodAcceleration

The *DoTodAcceleration* function handles the special case when there is an acceleration to the descent Mach at the top-of-descent. This function is invoked from *Add Descent Mach Waypoint*, which passes in the index number for the TOD waypoint, *TodIndex*, and the Mach value at the TOD, *MachAtTod.* The function will insert the Mach acceleration point into the waypoint list if a valid acceleration point can be found.

Make an initial estimate of the distance to the new Mach value. The function *TodAccelerationDistance* returns the values *Valid, k*, and *dx*.

TodAccelerationDistance(TodIdx, MachAtTod, Mach Descent Mach, Valid, k, dx)

if (Valid) then

Add the VTCP for the end of the TOD acceleration.

$d = DTG_{TodIdx} - dx$

The original ground track will be needed for the new TCP, so save it.

OldGroundTrack = GetTrajGndTrk(d)

Save the wind data at this distance as a temporary TCP.

GenerateWptWindProfile(d, TemporaryTcp)

The new waypoint is downstream of the current value of *k*.

$k = k + 1$

InsertTcp(k)

Note that TCP_k is the newly created TCP.

$TcpType_k = VTCP$

$TurnType_k = NO\ TURN$

If the new waypoint is not already marked as a special vertical type, mark it as a top-of-descent acceleration point.

if $(VSegType_k = NONE)$ $VSegType_k = TOD\ ACCELERATION$

$DTG_k = d$

Calculate the altitude for the new TCP.

$Altitude_k = Altitude_{TodIdx} - (NmiToFeet * dx) * tangent(Crossing\ Angle_{k+1})$

$Mach_k = Mach\ Descent\ Mach$

$Mach\ Cross_k = Mach\ Descent\ Mach$

$Mach\ Segment_k = true$

Set the *Crossing Rate* to the default value of 0.75.

$Crossing\ Rate_k = 0.75\ kt/sec$

Add the appropriate ground track value.

if $(WptInTurn(k))$ $Ground\ Track_k = ComputeGndTrk(k, d)$

else Ground Track$_k$ = OldGroundTrack

Copy the wind data from *TemporaryTcp* into *TCP$_k$*.

end of if (Valid)

else mark this as an error for being unable to accelerate to the descent Mach value. Note that this is not a fatal error.

EstimateNextCas

EstimateNextCas is an iterative function to estimate the CAS value, *CAS*, at the next TCP. Note that there is no closed-form solution for this calculation of CAS. The input variable names described in this function are from the calling routine and are, in order, the target CAS value, *Current CAS*; the ground speed at the estimation starting point, *Current Ground Speed*; an estimation limiting flag, *No Limit Flag*; the CAS at the estimation starting point, *Prior CAS*; the head wind at the estimation starting point, *Head Wind*; the altitude at the estimation starting point, *Altitude*; the distance from the estimation starting point to the point where the CAS is to be estimated, *Distance*; the deceleration rate to be used in this estimation, *CAS Rate;* and the temperature deviation at the end point, *Td*. Also, the input deceleration value must be greater than 0, *CAS Rate* > 0. The function returns the estimated CAS value.

Guess CAS = Current CAS

Set up a condition to get at least one pass.

*d = -10 * Distance*

*size = 1.01 * (Prior CAS - Guess CAS)*

count = 0

if ((Distance > 0) and (CAS Rate > 0)) then

Iterate a solution. The counter count is used to terminate the iteration if the distance estimation does reach a solution within 0.001 nmi.

while ((|Distance - d| > 0.001) and (count < 10))

if (Distance > d) Guess CAS = Guess CAS - size

else Guess CAS = Guess CAS + size

size = size / 2

The estimated time t to reach this speed,

t = (Guess CAS - Current CAS) / CAS Rate

The new ground speed,

Gs2 = CasToTas Conversion(Guess CAS, Altitude, Td) - Head Wind

*d = ((Current Ground Speed + Gs2) / 2) * (t / (3600 sec/hr))*

count = count + 1

end of the while loop

Limit the computed *CAS*, if necessary.

if ((NoLimit = false) and (Guess CAS > Prior CAS)) Guess CAS = Prior CAS

return Guess CAS

EstimateNextMach

EstimateNextMach is an iterative function to estimate the Mach value, *Mach*, at the next TCP. Note that there is no closed-form solution for this calculation of value. The input variable names described in this function are from the calling routine and are, in order, the target Mach value, *Current Mach*; the ground speed at the estimation starting point, *Current Ground Speed*; an estimation limiting flag, *No Limit Flag*; the Mach at the estimation starting point, *Prior Mach*; the head wind at the estimation starting point, *Head Wind*; the altitude at the estimation starting point, *Altitude*; the distance from the estimation starting point to the point where the Mach is to be estimated, *Distance*; the deceleration rate, in Mach, to be used in this estimation, *Mach Rate;* and the temperature deviation at the end point, *Td*. Also, the input deceleration value must be greater than 0, *Mach Rate* > 0. The function returns the estimated Mach value.

Mach = Current Mach

Set up a condition to get at least one pass.

*d = -10 * dx*

*size = 1.01 * (Prior Mach - Current Mach)*

count = 0

if ((dx > 0) and (Mach Rate > 0)) then

Iterate a solution. The counter count is used to terminate the iteration if the distance estimation does reach a solution within 0.001 nmi.

while ((|d - dx| > 0.001) and (count < 10))

if (d > dx) Mach = Mach - size

else Mach = Mach + size

size = size / 2

The estimated time t to reach this speed,

t = (Mach - Current Mach) / Mach Rate

The new ground speed,

CAS = MachToCas(Mach, Altitude)

Gs2 = CasToTas Conversion(CAS, Altitude, Td) - Head Wind

*d = ((Current Ground Speed + Gs2) / 2) * (t / (3600 sec/hr))*

count = count + 1

end of the while loop

Limit the computed *Mach*, if necessary.

if (Mach > Prior Mach) Mach = Prior Mach

return Mach

FindNextLDRWaypoint

This routine is used in the linear deceleration rate calculations to return the waypoint index number for the first linear deceleration waypoint where the deceleration value has not been calculated. If no such waypoint exists, the routine returns a value of -1.

index = -1

i = index number of first TCP

found = false

Find the index for the next linear deceleration value that hasn't been evaluated.

while ((i < index number of last TCP) and (found = false))

if ((LDR Flag$_i$ = true) and (LDR Finished$_i$ = false) and (LDR Pass Count$_i$ ≤ 4)) then

found = true

index = i

else i = i + 1

end of while ((i < index number of last TCP) and (found = false))

return index

GenerateWptWindProfile

The function *GenerateWptWindProfile* is used to compute new wind profile data. This function is a double-linear interpolation using the wind data from the two bounding input waypoints to compute the wind profile for a new VTCP, TCP_k. The interpolations are between the wind altitudes from the input data and

the ratio of the distance *d* at a point between TCP_{i-1} and TCP_i and the distance between TCP_{i-1} and TCP_i. E.g.,

– Find the two bounding input waypoints, TCP_{i-1} and TCP_i, between which *d* lies, e.g., $TCP_{i-1} \geq d \geq TCP_i$.

– Using the altitudes from the wind profile of TCP_i, compute and temporarily save the wind data at these altitudes using the wind data from TCP_{i-1} (e.g., $Wind\ Speed_{Temporary,\ Altitude1}$).

– Compute the wind speed, wind direction, and temperature deviation for each altitude using the ratio *r* of the distances. Assuming that the difference between DTG_{i-1} and $DTG_i \neq 0$, and that $DTG_{i-1} > DTG_i$.

$r = (DTG_{i-1} - d) / (DTG_{i-1} - DTG_i)$

Iterate the following for each altitude in the profile.

$Wind\ Speed_{k,\ Altitude1} = (1 - r) * Wind\ Speed_{Temporary,\ Altitude1} + r * Wind\ Speed_{i,\ Altitude1}$

$a = DeltaAngle(Wind\ Direction_{Temporary,\ Altitude1},\ Wind\ Direction_{i,\ Altitude1})$

$Wind\ Direction_{k,\ Altitude1} = Wind\ Direction_{k,\ Altitude1} + (r * a)$

$Temperature\ Deviation_{k,\ Altitude1} =$
$(1 - r) * Temperature\ Deviation_{Temporary,\ Altitude1} + r * Temperature\ Deviation_{i,\ Altitude1}$

Figure 11 is an example of the computation data for the wind computation at a 9,000 ft altitude. In this example, TCP_{i-1} has wind data at 10,000 and 8,000 ft and TCP_i has wind data at 9,000 ft.

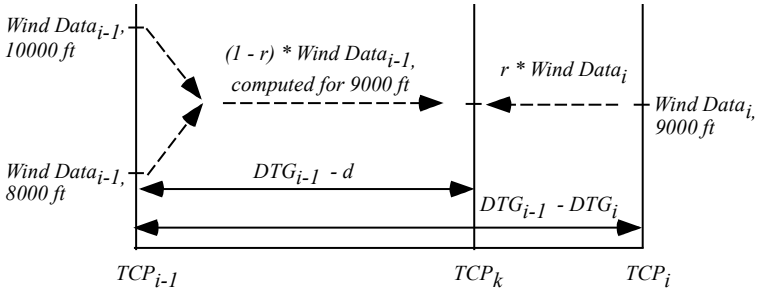


Figure 11. Example of computing a single wind data altitude.

GetTrajGndTrk

The *GetTrajGndTrk* function computes the ground track at the along-path distance, *distance*.

If the distance value is out of range of the trajectory, just return the ground track at the beginning or end of the trajectory.

if (distance < 0) Ground Track = Ground Track$_{last\ TCP}$

else if (distance > DTG$_{first\ TCP}$*) Ground Track = Ground Track*$_{first\ TCP}$

else

Find where this distance is on the path.

i = index number of the last TCP

while (distance > DTG_i) i = i -1

if (distance = DTG_i) Ground Track = $Ground\ Track_i$

else

x = DTGi - DTG_{i+1}

if (x ≤ 0) r = 0

else r = (distance - DTG_{i+1}) / x

if (r > 1) r = 1

*dx = (1 - r) * DeltaAngle($Ground\ Track_i$, $Ground\ Track_{i+1}$)*

Ground Track = $Ground\ Track_i$ + dx

return Ground Track

ComputeDescentAccelDecel

The function *ComputeDescentAccelDecel* is designed to handle the special case of a Mach acceleration in the descent where the first CAS crossing restriction cannot be met. The calling program provides as input and retains the subsequent outputs for the following variables: *CasIndex, CruiseMach, MachCasModified, DescentMach,* and *MachCas*. The variable *CasIndex* is the index value in the TCP list for the first CAS constrained waypoint. The variable *CruiseMach* is the last Mach crossing restriction value prior to the first CAS segment. The variable *MachCasModified* is a flag returned by this function if the *DescentMach* or *MachCas* values are changed. The variables *DescentMach* and *MachCas* are the planned descent Mach and planned Mach-to-CAS transition CAS, respectively, and these values may be modified by this function.

Initialize variables.

i = 0

z = 0

fini = false

MachCasModified = false

Perform up to two iterations to calculate any required Mach or CAS change in the descent.

while ((fini = false) and (i < 2))

Calculate *z* at the descent Mach and the Mach-to-CAS CAS.

z = FindMachCasTransitionAltitude(MachCas, DescentMach)

Determine if *z* is below the CAS crossing restriction.

if (z < $Altitude_{CasIndex}$)

Set the CAS to the value at this altitude, knowing the crossing restriction can't be met.

MachCas = MachToCas(DescentMach, $Altitude_{CasIndex}$)

else if (z > $Altitude\ Cross_{first\ TCP}$)

Set the Mach to the descent CAS at the cruise altitude.

m = CasToMach(MachCas, $Altitude_{first\ TCP}$)

if (m > CruiseMach) DescentMach = m

if (MachCas <$Crossing\ CAS_{CasIndex}$) then

MachCas = $Crossing\ CAS_{CasIndex}$

i = i + 1

else fini = true

end of while ((fini = false) and (i < 2))

Find the TOD TCP.

fini = false

TodIndex = 0

i = index number of the first TCP

while ((i < index number of the last TCP) and (fini = false))

if (($Altitude_i$ < $Altitude_{first\ TCP}$) or ($Crossing\ CAS_i$ > 0)) then

if (($Altitude_i$ ≠ $Altitude_{first\ TCP}$)) TodIndex = i - 1

else TodIndex = i

fini = true

i = i + 1

end of while ((i < index number of the last TCP) and (fini = false))

Calculate the entire decent distance.

$d = DTG_{TodIndex} - DTG_{CasIndex}$

Estimate the distance, *Daccel,* to the new Mach value.

TodAccelerationDistance(TodIndex, CruiseMach, MachDescentMach, Valid, AccelIndex, Daccel)

Estimate the distance, *Ddecel,* to the CAS crossing speed.

BodDecelerationDistance(CasIndex, z, Mach Transition CAS, Ddecel)

fini = false

m = DescentMach

The nominal speed values won't work, there is insufficient distance to obtain the acceleration and then slow to the crossing speed. Iterate until a solution is found.

while ((fini = false) and (d < (Daccel + Ddecel)))

Iterate the solution.

Slightly change the Mach and then find the CAS.

m = m - 0.002

if (m < Cruise Mach) then

m = Cruise Mach

fini = true

Estimate the distance to the new Mach value.

TodAccelerationDistance(TodIndex, Cruise Mach, m, Valid, AccelIndex, Daccel)

Find the altitude where the acceleration ends.

$z = Crossing\ Altitude_{first\ TCP} - (Daccel / d) * (Crossing\ Altitude_{first\ TCP} - Crossing\ Altitude_{CasIndex})$

CAS = MachToCas(m, z)

Estimate the distance to the CAS crossing speed.

BodDecelerationDistance(CasIndex, z, CAS, Ddecel)

if (d ≥ (Daccel + Ddecel)) then

fini = true

Modify the descent Mach and CAS values.

modified = true

DescentMach = m

Add a buffer to the CAS so that subsequent Mach-to-CAS calculation won't cause an error.

MachCas = CAS + 0.1

end of if (d ≥ (Daccel + Ddecel))

GetWindAtAltitudeDistance

The function *GetWindAtAltitudeDistance* is used to compute the wind speed, wind direction, and temperature deviation at an altitude, *Altitude*, for a specific distance, *Distance*, along the path. This function is a linear interpolation using the wind data from the input waypoints that bound the along-path distance.

Find the bounding input waypoints.

i0 = index number of the first TCP

j = index number of the first TCP

fini = false

if (Distance < 0) Distance = 0

while ((fini = false) and (j < index number of the last TCP))

if ((TcpType$_j$ = INPUT) and (DTG$_j$ ≥ Distance)) i0 = j

if (DTG$_j$ < Distance) fini = true

j = j + 1

end of the while loop

i1 = i0 + 1

j = i1

fini = false

while ((fini = false) and (j < index number of the last TCP))

if ((TcpType$_j$ = INPUT) and (DTG$_j$ ≤ Distance)) then

i1 = j

fini = true

end of if

j = j + 1

end of the while loop

if (i1 > index number of the last TCP) i1 = index number of the last TCP

if (i0 = i1) InterpolateWindWptAltitude(TCP_{i0}, Altitude, Ws, Wd, Td)

else

Interpolate the winds at each waypoint.

InterpolateWindWptAltitude(TCP_{i0}, Altitude, Spd0, Dir0, Td0)

InterpolateWindWptAltitude(TCP_{i1}, Altitude, Spd1, Dir1, Td1)

Interpolate the winds between the two waypoints.

InterpolateWindAtRange(Distance, DTG_{i0}, Spd0, Dir0, Td0, DTG_{i1}, Spd1, Dir1, Td1, Wind Speed, Wind Speed, Temperature Deviation)

InterpolateWindAtRange

The function *InterpolateWindAtRange* is used to compute the wind speed, *Wind Speed,* wind direction, *Wind Direction,* and temperature deviation, *Temperature Deviation,* at a distance along path, *Distance*, between two sets of wind data sets, denoted by the subscripts *1* and *2*, where $DTG_1 \geq Distance \geq DTG_2$. This function is a linear interpolation using the wind data from the input.

if (($DTG_1 = DTG_2$) or ((Distance = DTG_1)) then

Wind Speed = $WindSpd_1$

Wind Direction = $WindDir_1$

Temperature Deviation = $TempDev_1$

else if (Distance = DTG_2) then

Wind Speed = $WindSpd_2$

Wind Direction = $WindDir_2$

Temperature Deviation = $TempDev_2$

else

Interpolate the values.

$r = (DTG_1 - Distance) / (DTG_1 - DTG_2)$

$Wind\ Speed = (1 - r) * WindSpd_1) + (r * WindSpd_2)$

$a = DeltaAngle(WindDir_1, WindDir_2)$

$Wind\ Direction = WindDir_1 + (r * a)$

$AdjustAngle(Wind\ Direction)$

$Temperature\ Deviation = ((1 - r) * TempDev_1) + (r * TempDev_2)$

InterpolateWindWptAltitude

The function *InterpolateWindWptAltitude* is used to compute the wind speed, *Wind Speed*, wind direction, *Wind Direction*, and temperature deviation, *Temperature Deviation*, at an altitude, *Altitude*, for TCP_i. This function is a linear interpolation using the wind data from the TPC_i.

Find the index numbers, *p0* and *p1*, for the bounding altitudes.

$p0 = 0$

$p1 = 0$

$for\ (k = 1;\ k \leq Number\ of\ Wind\ Altitudes_i;\ k = k + 1)$

$if\ (Wind\ Altitude_{i,\ k} \leq Altitude)\ p0 = k$

$if\ ((Wind\ Altitude_{i,\ k} \geq Altitude) and\ (p1 = 0))\ p1 = k$

$if\ (p1 = 0)\ p1 = Number\ of\ Wind\ Altitudes_i$

$if\ (p0 = p1)\ then$

$Wind\ Speed = Wind\ Speed_{p0}$

$Wind\ Direction = Wind\ Direction_{p0}$

$Temperature\ Deviation = Temperature\ Deviation_{p0}$

$else$

$if\ (Wind\ Altitude_{p1} \leq Wind\ Altitude_{p0})\ r = 0$

$else\ r = (Altitude - Wind\ Altitude_{p0}) / (Wind\ Altitude_{p1} - Wind\ Altitude_{p0})$

$Wind\ Speed = ((1 - r) * Wind\ Speed_{p0}) + (r * Wind\ Speed_{p1})$

$a = DeltaAngle(Wind\ Direction_{p0},\ Wind\ Direction_{p1})$

$Wind\ Direction = Wind\ Direction_{p0} + (r * a)$

AdjustAngle(Wind Direction)

*Temperature Deviation = ((1 - r) * Temperature Deviation$_{p0}$) + (r * Temperature Deviation$_{p1}$)*

FindAltitude

The function *FindAltitude* is used to compute the path altitude at a specific distance, *Dtg*, along the trajectory.

alt = 0

fini = false

found = false

done = false

i = index number of first TCP

while (fini = false)

if (DTG$_i$ == Dtg) then

alt = Altitude$_i$

done = true

fini = true

else

if (DTG$_i$ > Dtg) i = i + 1

else

fini = true

found = true

if (i ≥ index number of last TCP) fini = true

end of while (fini = false)

if (done = false)

The distance is not at a TCP. Calculate the altitude along the segment.

if (found = false) alt = Altitude$_{last\ TCP-1}$

else

d = DTG$_{i-1}$ - DTG$_i$

if ($d \leq 0.0$) $alt = Altitude_{i-1}$

else

$alt = (dtg - DTG_i) / d * (Altitude_{i-1} - Altitude_i) + Altitude_i$

end of if (done = false)

return alt

FindMachCasTransitionAltitude

The function *FindMachCasTransitionAltitude* is used to compute the altitude where the input Mach, *Mach*, and CAS, *Cas*, values would be equivalent.

$$z = (1 - (((((0.2 * ((Cas/661.48)^2) + 1)^{3.5}) - 1) / (((0.2 * (Mach^2) + 1)^{3.5}) - 1))^{0.19026})) / 0.00000687535$$

LDRAddRate

The function *LDRAddRate* is used in the linear deceleration rate calculations to estimate a new deceleration rate for the case where the end speed is reached after the speed-restricted waypoint; i.e., a higher deceleration is required. The waypoint index variable *idx* is used to identify the starting TCP number in these determinations.

If the crossing rate is equal to or greater than a maximum crossing rate, 5 kt/sec in this implementation, then no further rate calculations will be performed on this TCP.

if ($Crossing\ Rate_{idx} \geq 5$ *kt/sec) LDR Finished = true*

else

Calculate a higher deceleration value.

$delta = LDR\ Last\ Rate_{idx} - Crossing\ Rate_{idx}$

$Crossing\ Rate_{idx} = Crossing\ Rate_{idx} + 0.05 * delta$

if ($Crossing\ Rate_{idx} \geq 5$ *kt/sec)* $Crossing\ Rate_{idx} = 5$ *kt/sec*

$LDR\ Pass\ Count_{idx} = LDR\ Pass\ Count_{idx} + 1$

LDRFindAcquiredSpeed

The function *LDRFindAcquiredSpeed* is used in the linear deceleration rate calculations to find the index value for a prior waypoint with a speed value that is equal to or greater than *Match Speed*. In addition to the *Match Spee*d input value, the waypoint index variable *idx* is used to identify the starting waypoint number in this search.

Find the previous speed in the TCP list.

ObtainedIndex = index number of first TCP

i = idx - 1

found = false

while ((i ≥ index number of first TCP) and (found = false))

if ((Mach Segment$_{idx}$ = false) and (Mach Segment$_i$ = true)) then

fini = false

j = idx

while ((j ≥ index number of first TCP) and (fini = false))

if (Mach Segment$_j$ = true) fini = true

else j = j - 1

spd = CAS$_j$

end of if ((Mach Segment$_{idx}$ = false) and (Mach Segment$_i$ = true))

else if (Mach Segment$_{idx}$ = true) spd = Mach$_i$

else spd = CAS$_i$

Determine if the current test speed matches or exceeds the value of *Match Speed*.

if (spd ≥ MatchSpeed) then

found = true

ObtainedIndex = i

i = i - 1

end of while ((i ≥ index number of first TCP) and (found = false))

return the value of *ObtainedIndex*.

LDRFindLastSpeedConstraint

The function *LDRFindLastSpeedConstraint* is used in the linear deceleration rate calculations to find the index value for the speed constraint prior to the waypoint with the planned linear deceleration. The waypoint index variable *idx* is used to identify the starting waypoint number in this search.

LastIndex = index number of first TCP

i = idx - 1

found = false

while ((i ≥ index number of first TCP) and (found = false))

if ((TcpType$_i$ = INPUT) and ((Cas Cross$_i$ > 0) or (Mach Cross$_i$ > 0))) then

LastIndex = i

found = true

i = i - 1

return LastIndex

LDRRateCheck

The function *LDRRateCheck* is used in the linear deceleration rate calculations to determine if the current, estimated deceleration rate is too small to meet the deceleration requirement. It also will report a non-LDR speed error if one exists. This function will return a value of true for the variable *LdrRateError* if the current linear deceleration segment is not valid and will return a value of true for the variable *OtherError* if a valid speed could not be calculated for any segment. The waypoint index variable *idx* is used to identify the starting TCP number in these determinations and the *LDRPending* flag is used to identify the special case when the LDR calculations are being initialized.

LdrRateError = false

skip = false

Do not begin processing LDR waypoints if the basic trajectory is invalid.

if ((LDRPending = true) and (Fatal Error = true)) skip = true,

referring to the section *Errors in the Trajectory* regarding marked error conditions.

if (skip = false) then

if (Fatal Error = true) then

Determine if the error is for the current deceleration point.

Find the begin-deceleration waypoint.

baseidx = index number of the first TCP

i = index number of the first TCP

found = false

while ((i ≤ index number of the last TCP) and (found = false))

if ((DTG$_i$ = LDR Base Distance$_{idx}$) and (TcpType$_i$ = INPUT)) then

baseidx = i

found = true

i = i + 1

end of while ((i ≤ index number of the last TCP) and (found = false))

Find the LDR waypoint with an error speed-related error.

i = index number of the first TCP

while ((i ≤ index number of the last TCP) and (LdrRateError = false))

if ((i ≥ baseidx) and (i ≤ idx) and ($Error_i$ = any speed error)) LdrRateError = true

i = i + 1

end of if (Fatal Error = true) then

Now determine if there are other speed errors.

OtherError = false

i = index number of the first TCP

while ((i ≤ index number of the last TCP) and (OtherError = false))

if ($Error_i$ = any speed error) OtherError = true

i = i + 1

end of if (skip = false) then

else OtherError = true

LDReduceRate

The function *LDReduceRate* is used in the linear deceleration rate calculations to estimate a new deceleration rate for the case where the end speed is reached before the speed-restricted waypoint; i.e., a lower deceleration is required. The waypoint index variable *idx* is used to identify the starting TCP number in these determinations.

Calculate the distance between the deceleration points.

d1 = LDR Base Distance$_{idx}$ - DTG$_{idx}$

Calculate the distance to the current deceleration point.

d2 = LDR Obtained Distance$_{idx}$ - DTG$_{idx}$

if (d1 > 0) d = 1 - d2 / d1

else d = 1

Set the finish flag to true if the calculated deceleration has resulted in a 'close enough' result. In this implementation, 'close enough' is 500 ft, 0.0822909809 nmi.

if (((d1 - d2) < 0.0822909809*) or (d = 1)) LDR Finished$_{idx}$ = true*

else

LDR Last Rate$_{idx}$ = Crossing Rate$_{idx}$

*Crossing Rate$_{idx}$ = Crossing Rate$_{idx}$ - d * LDR Last Rate$_{idx}$*

Do not allow the deceleration rate to be less than 0.01 kt/sec.

if (Cross Rate$_{idx}$ < 0.01 kt/sec) Crossing Rate$_{idx}$ = 0.01 kt/sec

LDR Last Ratio$_{idx}$ = d

LDR Pass Count$_{idx}$ = LDR Pass Count$_{idx}$ + 1

RadialRadialIntercept

The function *RadialRadialIntercept* determines if two place-and-radial sets, each defined by latitude, longitude, and a track angle, will intersect and if so, calculates the latitude and longitude of the intercept point. Inputs are values of latitude, *Latitude*, longitude, *Longitude*, and angle*, Angle*; one set of each for the two place-and-radial sets. If a valid intercept can be calculated, then the intercept point's latitude and longitude are output, *NewLatitude* and *NewLongitude*, and the function returns a valid indication. Otherwise, the function returns an invalid indication.

Calculate the distance and the track angle between the two input positions.

*distance$_{1,2}$ = arccosine(sine(Latitude$_1$) * sine(Latitude$_2$) + cosine(Latitude$_1$) * cosine(Latitude$_2$) * cosine(Longitude$_1$ - Longitude$_2$))*

*track$_{1,2}$ = arctangent2(sine(Longitude$_2$ - Longitude$_1$) * cosine(Latitude$_2$), cosine(Latitude$_1$) * sine(Latitude$_2$) - sine(Latitude$_1$) * cosine(Latitude$_2$) * cosine(Longitude$_2$ - Longitude$_1$))*

Check for error in the intercept calculation.

error = false

track$_1$ = Angle$_1$ - track$_{1,2}$ + 90°

Adjust *track$_1$* such that 0° ≥ *track$_1$* ≥ 360°.

AdjustAngle(track$_1$)

track$_2$ = Angle$_2$ - track$_{1,2}$ + 90°

Adjust *track$_2$* such that 0° ≥ *track$_2$* ≥ 360°.

AdjustAngle(track$_2$)

Determine the quadrant.

$ang_1 = track_2 + 180°$

Adjust ang_1 such that $0° \geq ang_1 \geq 360°$.

AdjustAngle(ang_1)

if ((|DeltaAngle(track1 , track2)| < 2°) or (|DeltaAngle(track1 , ang1)| < 2°)) then

Determine if the angles are really 180 degrees apart.

$ang_2 = Angle_2 + 180°$

Adjust ang_2 such that $0° \geq ang_2 \geq 360°$.

AdjustAngle(ang_2)

$ang_3 = DeltaAngle(Angle_1, ang_2)$

$ang_4 = DeltaAngle(Angle_1, track_{1,2})$

if ((|ang3| > 2°) or (|ang4| > 2°)) error = true

if (error = false)

$RelativeLatLong(Latitude_1, Longitude_1, track_{1,2}, distance_{1,2} / 2, NewLatitude, NewLongitude)$

else

Determine the quadrant.

$if\ (track_1 \leq 90°)\ quadrant1 = 1$

$else\ if\ (track_1 \leq 180°)\ quadrant1 = 2$

$else\ if\ (track_1 \leq 270°)\ quadrant1 = 3$

else quadrant1 = 4

$if\ (track_2 \leq 90°)\ quadrant2 = 1$

$else\ if\ (track_2 \leq 180°)\ quadrant2 = 2$

$else\ if\ (track_2 \leq 270°)\ quadrant2 = 3$

else quadrant2 = 4

if (quadrant1 = 1) then

if ((quadrant2 = 2) or (quadrant2 = 3)) error = true

if ((quadrant2 = 1) and (chktk1 < chktk2)) error = true

else if (quadrant1 = 2) then

if ((quadrant2 = 1) or (quadrant2 = 4)) error = true

if ((quadrant2 = 2) and (chktk1 > chktk2)) error = true

else if (quadrant1 = 3) then

if ((quadrant2 = 1) or (quadrant2 = 2) or (quadrant2 = 4)) error = true

if ($track_1$ > $track_2$) error = true

else

if ((quadrant2 = 1) or (quadrant2 = 2) or (quadrant2 = 3)) error = true

if ($track_1$ < $track_2$) error = true

if (error = false) then

trx_1 = |$Angle_1$ - $track_{1,2}$|

Adjust trx_1 such that 0° ≥ trx_1 ≥ 360°

trx_2 = |$Angle_2$ - ($track_{1,2}$ + 180°)|

Adjust trx_2 such that 0° ≥ trx_2 ≥ 360°

if (trx_1 > 180°) trx_1 = 360° - trx_1

if (trx_2 > 180°) trx_2 = 360° - trx_2

ang_5 = 180° - trx_1 - trx_2

if ((ang_5 = 0°) or ((ang_5-180°) = 0°) or ($distance_{1,2}$ = 0)) error = true

if (error = false) then

*$distance_2$ = $distance_{1,2}$ * sine(trx_2) / sine(ang_5)*

if ($distance_2$ ≤ 0) $distance_2$ = - $distance_2$

if ($distance_2$ > max_intercept_range) error = true

else RelativeLatLong($Latitude_1$, $Longitude_1$, $Angle_1$, $distance_2$, NewLatitude, NewLongitude)

if (error) return false

else return true

RelativeLatLon

The function *RelativeLatLon* computes the latitude and longitude from input values of latitude, *BaseLat*, longitude, *BaseLon*, angle, *Angle*, and range, *Range*.

DegreesToNmi = 60.0405

if (Angle = 180°) Latitude = -Range / DegreesToNmi + BaseLat

*else Latitude = ((Range * cos(Angle)) / DegreesToNmi) + BaseLat*

if ((BaseLat = 0) or (BaseLat = 180)) Longitude = BaseLon

*else if (Angle = 90°) Longitude = BaseLon + Range / (DegreesToNmi * cos(BaseLat))*

*else if (Angle = 270°) Longitude = BaseLon - Range / (DegreesToNmi * cos(BaseLat))*

else

*r1 = tangent(45° + 0.5 * Latitude)*

r2 = tangent(4°5 + 0.5 BaseLat)*

if ((r1 = 0) or (r2 = 0)) Longitude = 20, just some number, mark this as a fatal error condition.

*else Longitude = BaseLon + (180° / pi *(tangent(Angle)* (log(r1) - log(r2))))*

TodAccelerationDistance

The *TodAccelerationDistance* function estimates the distance required for the special case of an acceleration from the top-of-descent Mach to the descent Mach at the top-of-descent. This function is invoked from *ComputeDescentAccelDecel* and *DoTodAcceleration*, which passes in the index number for the TOD waypoint, *TodIndex*, and the Mach value at the TOD, *MachAtTod.* The function returns a validity flag to indicate if a TOD acceleration is valid, *Valid*, and if valid, the indices in the TCP list where the acceleration occurs, *AccelIndex*, and the distance from the index point of the acceleration, *Distance*.

Perform an initialization of flags and counters.

skip = true

k = TodIndex

Make an initial guess of the distance to the new Mach value using a 0.75 kt/sec acceleration value.

$Mach\ Rate_1 = CasToMach(0.75\ kt / sec, Altitude_{TodIndex})$

Compute the time required to do the deceleration.

t = (Mach Descent Mach – MachAtTod) / Mach Rate$_1$

Compute the wind speed and direction at the current altitude.

InterpolateWindWptAltitude(Wind Profile$_{TodIndex}$, Altitude$_{TodIndex}$, Wind Speed, Wind Direction, Temperature Deviation)

Get the ground track at the current point.

if (WptInTurn(Waypoint$_{TodIndex}$)) track = Ground Track$_{TodIndex + 1}$

else track = Ground Track$_{TodIndex}$

TOD Ground Speed = ComputeGndSpeedUsingMachAndTrack(MachAtTod, track, Altitude$_{TodIndex}$, Wind Speed, Wind Direction, Temperature Deviation)

Descent Ground Speed = ComputeGndSpeedUsingMachAndTrack(Mach Descent Mach, track, Altitude$_{TodIndex}$, Wind Speed, Wind Direction, Temperature Deviation)

The average ground speed is as follows:

Average Ground Speed = (TOD Ground Speed + Descent Ground Speed) / 2

The distance estimate, dx, is *Average Ground Speed* * t with a conversion to nmi.

dx = Average Ground Speed * t / (3600 sec/hr)

Now compute better estimates, doing this twice to refine the estimation.

for ($i = 1$; $i \leq 2$; $i = i + 1$)

skip = false

Determine if this distance is beyond the next downstream waypoint.

k = TodIndex

$d = DTG_{TodIndex} - dx$

while (($k <$ (index number of the last TCP – 1)) and ($DTG_{k+1} > d$))

if (($k \neq$ TodIndex) and (Crossing Rate$_k > 0$)) skip = true

$k = k + 1$

Compute the wind speed and direction at the new altitude.

InterpolateWindWptAltitude(Waypoint$_k$, Altitude$_k$, Wind Speed, Wind Direction, Temperature Deviation)

The ground speed at this point is:

Descent Ground Speed = ComputeGndSpeedUsingMachAndTrack(Mach Descent Mach, Ground Track$_k$, Altitude$_k$, Wind Speed, Wind Direction, Temperature Deviation)

The average ground speed is:

Average Ground Speed = (TOD Ground Speed + Descent Ground Speed) / 2

The distance, *dx*, is:

*dx = Average Ground Speed * t / (3600 sec/hr)*

end of for (i = 1; i ≤ 2; i = i + 1)

If there is a valid deceleration point, add it.

if (skip = false) Valid = true

else Valid = false

AccelIndex = k

Distance = dx

TodDecelerationDistance

The *TodDecelerationDistance* function estimates the distance required for the special case of n deceleration from the top-of-descent Mach to the descent CAS at the top-of-descent. This function is invoked from *Add TOD Deceleration TCP*, which passes in the index number for the TOD waypoint, *TodIndex*, the Mach value at the TOD, *MachAtTod*, and the descent CAS, *DescentCas*. The function returns a validity flag to indicate if a TOD deceleration is valid, *Valid*, and if valid, the indices in the TCP list where the deceleration occurs, *DecelIndex*, and the distance from the index point of the deceleration, *Distance*.

Perform an initialization of flags and counters.

skip = true

k = TodIndex

dx = 0

Estimate the distance to the new CAS value.

TodCas = MachToCas(MachAtTod, Altitude$_{TodIdx}$)

if (TodCas > DescentCas) then

Calculate the time to do the deceleration using the default Mach-to-CAS deceleration of 0.25 kt/sec.

t = (TodCas - DescentCas) / 0.25 kt/sec

Compute the wind speed, direction, and temperature deviation at the current altitude.

InterpolateWindWptAltitude(TodIdx, Altitude$_{TodIdx}$, Ws, Wd, TempDev)

Save the ground track value at the current point.

if (WptInTurn(TodIdx)) track = Ground Track$_{TodIdx+1}$

else track = Ground Track$_{TodIdx}$

TodGs = ComputeGndSpeedUsingMachAndTrack(MachAtTod, track, Altitude$_{TodIdx}$, Ws, Wd, TempDev)

The altitude for the deceleration endpoint is unknown; estimate the ground speed using the TOD altitude.

DescentGs = ComputeGndSpeedUsingTrack(DescentCas, track, Altitude$_{TodIdx}$, Ws, Wd, Td)

Calculated the average groundspeed *AvgGs.*

AvgGs = (TodGs + DescentGs) / 2

The distance estimate, *dx,* is *AvgGs * t.*

*dx = AvgGs * t / (3600 sec/hr)*

Now compute better estimates, doing this twice to refine the estimation.

for (i = 1; i ≤ 2; i = i + 1)

skip = false

Determine if this distance is beyond the next downstream TCP.

k = TodIdx

d = DTG$_{TodIdx}$ - dx

alt = FindAltitude(d)

track = GetTrajGndTrk(d)

Using a temporary wind profile data set, p, generate the wind speed, *Ws*, direction, *Wd*, and temperature deviation, *TempDev*, at this altitude.

GenerateWptWindProfile(d, p)

InterpolateWindWptAltitude(p, alt, Ws, Wd, TempDev)

Calculate the groundspeed at this point.

DescentGs = ComputeGndSpeedUsingTrack(DescentCas, track, alt, Ws, Wd, TempDev)

Again, calculated the average groundspeed *AvgGs*.

AvgGs = (TodGs + DescentGs) / 2

Again, the distance estimate, *dx,* is *AvgGs * t.*

*dx = AvgGs * t / (3600 sec/hr)*

end of for (i = 1; i ≤ 2; i = i + 1)

end of if (TodCas > DescentCas)

if (skip = false) Valid = true

else Valid = false

DecelIndex = k

Distance = dx

WptInTurn

The *WptInTurn* function simply determines if the waypoint is between a turn-entry TCP and a turn-exit TCP. If this is true, then the function returns a value of true, otherwise, it returns a value of false.

fini = false

within = false

j = i + 1

while ((fini = false) and (j < (index number of the last TCP)))

if ($TurnType_j$ = TURN START) fini = true

else if ($TurnType_j$ = TURN END)

fini = true

within = true

j = j + 1

return within

Summary

The algorithm described in this document takes as input a list of waypoints, their trajectory-specific data, and associated wind profile data. This algorithm calculates the altitude, speed, along path distance, and along path time for each waypoint and every point along the path where the speed, altitude, or ground track changes. A full 4D trajectory can then be generated by the techniques described. This documentation was based on an operational software prototype.

References

1. Abbott, T. S.; and Moen, G. C,: *Effect of Display Size on Utilization of Traffic Situation Display for Self-Spacing Task*, NASA TP-1885, 1981.
2. Abbott, Terence S.: *A Compensatory Algorithm for the Slow-Down Effect on Constant-Time-Separation Approaches*, NASA TM-4285, 1991.
3. Sorensen, J. A.; Hollister, W.; Burgess, M.; and Davis, D.: *Traffic Alert and Collision Avoidance System (TCAS) - Cockpit Display of Traffic Information (CDTI) Investigation*, DOT/FAA/RD-91/8, 1991.
4. Williams, D. H.: *Time-Based Self-Spacing Techniques Using Cockpit Display of Traffic Information During Approach to Landing in a Terminal Area Vectoring Environment*, NASA TM-84601, 1983.
5. Koenke, E.; and Abramson, P.: *DAG-TM Concept Element 11, Terminal Arrival: Self Spacing for Merging and In-trail Separation, Advanced Air Transportation Technologies Project*, 2004.
6. Abbott, T. S.: *Speed Control Law for Precision Terminal Area In-Trail Self Spacing*, NASA TM 2002-211742, 2002.
7. Osaguera-Lohr, R. M.; Lohr, G. W.; Abbott, T. S.; and Eischeid, T. M.: *Evaluation Of Operational Procedures For Using A Time-Based Airborne Interarrival Spacing Tool*, AIAA-2002-5824, 2002.
8. Lohr, G. W.; Osaguera-Lohr, R. M.; and Abbott, T. S.: *Flight Evaluation of a Time-based Airborne Inter-arrival Spacing Tool*, Paper 56, Proceedings of the 5th USA/Europe ATM Seminar at Budapest, Hungary, 2003.
9. Hoffman, E.; Ivanescu, D.; Shaw, C.; and Zeghal, K.: *Analysis of Constant Time Delay Airborne Spacing Between Aircraft of Mixed Types in Varying Wind Conditions*, Paper 77, Proceedings of the 5th USA/Europe ATM Seminar at Budapest, Hungary, 2003.
10. Ivanescu, D.; Powell, D.; Shaw, C.; Hoffman, E.; and Zeghal, K.: *Effect Of Aircraft Self-Merging In Sequence On An Airborne Collision Avoidance System*, AIAA 2004-4994, 2004.
11. Barmore, B. E.; Abbott, T. S.; and Krishnamurthy, K.: *Airborne-Managed Spacing in Multiple Arrival Streams*, Proceedings of the 24th Congress of the International Council of Aeronautical Sciences, 2004.
12. Oseguera-Lohr, R. M.; and Nadler, E. D.: *Effects of an Approach Spacing Flight Deck Tool on Pilot Eyescan*, NASA/TM-2004-212987, 2004.
13. Lohr, G. W.; Oseguera-Lohr, R. M.; Abbott, T. S.; and Capron, W. R.: *A Time-Based Airborne Inter-Arrival Spacing Tool: Flight Evaluation Result*, ATC Quarterly, Vol 13 no 2, 2005.
14. Krishnamurthy, K.; Barmore, B.; and Bussink, F. J. L.: *Airborne Precision Spacing in Merging Terminal Arrival Routes: A Fast-time Simulation Study*, Proceedings of the 6th USA/Europe ATM Seminar, 2005.
15. Weitz, L.; Hurtado, J. E.; and Bussink, F. J. L.: *Increasing Runway Capacity for Continuous Descent Approaches Through Airborne Precision Spacing*, AIAA 2005-6142, 2005.
16. Krishnamurthy, K.; Barmore, B.; Bussink, F. J.; Weitz, L.; and Dahlene, L.: *Fast-Time Evaluations Of Airborne Merging and Spacing In Terminal Arrival Operations*, AIAA-2005-6143, 2005.
17. Barmore, B.; Abbott, T. S.; and Capron, W. R.: *Evaluation of Airborne Precision spacing in a Human-in-the-Loop Experiment*, AIAA-2005-7402, 2005.
18. Hoffman, E., Pène, N., Rognin, L., Trzmiel, A., and Zeghal, K.: *Airborne Spacing: Managed vs. Selected Speed Mode on the Flight Deck*, 6th AIAA Aviation Technology, Integration and Operations Conference, 2006.
19. Baxley, B.; Barmore, B.; Bone, R.; and Abbott, T. S.: *Operational Concept for Flight Crews to Participate in Merging and Spacing of Aircraft*, 2006 AIAA Aviation Technology, Integration and Operations Conference, 2006.
20. Lohr, G. W.; Oseguera-Lohr, R. M.; Abbott, T. S.; Capron, W. R.; and Howell, C. T.: *Airborne Evaluation and Demonstration of a Time-Based Airborne Inter-Arrival Spacing Tool*, NASA/TM-2005-213772, 2005.

21. Barmore, B.; Abbott, T. S.; Capron, W. R.; and Baxley, B.: *Simulation Results for Airborne Precision Spacing along Continuous Descent Arrivals*, AIAA-2008-8931, 2008.

22. Swieringa , K. A.; Murdoch, J. L.; and Baxley, B.: *Evaluation of an Airborne Spacing Concept, On-board Spacing Tool, and Pilot Interface*, Proceedings of the 11th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference, 2011.

23. Weitz, L.A.; and Hurtado, J. E.: *Evaluation of an Airborne Spacing Concept, On-board Spacing Tool, and Pilot Interface*, Proceedings of the 11th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference, 2011.

24. Barmore, B.; Smith, C. L.; Palmer, S. O.; and Abbott, T. S.: *A Comparative Study of Interval Management Control Law Capabilities*, Proceedings of the IEEE/AIAA 31st Digital Avionics Systems Conference (DASC), 2012.

25. Barmore, B.; Baxley, B.; Abbott, T. S.; Capron, W. R.; Smith, C. L.; Shay, R. F.; and Hubbs, C.: *A Concept for Airborne Precision Spacing for Dependent Parallel Approaches*, NASA/TM-2012-217346, 2012.

26. Baxley, B.; Murdoch, J. L.; Swieringa , K. A.; Barmore, B.; Capron, W. R.; Hubbs, C.; Shay, R. F.; and Abbott, T. S.: *Experiment Description and Results for Arrival Operations Using Interval Management with Spacing to Parallel Dependent Runways (IMSPiDR)*, NASA/TP-2013-217998, 2013.

27. Weitz, L.A.: *Investigating String Stability of a Time-History Control Law for Interval Management*, Transportation Research Part C: Emerging Technologies 33, 2013.

28. Barmore, B.; Swieringa , K. A.; Underwood, M.; Abbott, T. S.; and Leonard, R. D.: *Development of an Interval Management Algorithm using Ground Speed Feedback for Delayed Traffic*, AIAA 2016-000754, 2016.

29. Swieringa , K. A.; Wilson , S. R.; Baxley, B.; Roper, R. D.; Abbott, T. S.; Levitt, I.; and Scharl, J.: *Flight Test Evaluation of the ATD-1 Interval Management Application*, AIAA 2017-4094, 2017.

30. Baxley, B.; Swieringa , K. A.; Wilson , S. R.; Roper, R. D.; Abbott, T. S.; Hubbs, C.; and Shay, R. F.: *Air Traffic Management Technology Demostration-1 (ATD-1) Avionics Phase 2 Flight Test and Results*, NASA/TP- 2018-219814, 2018.

31. Abbott, T. S.: *A Trajectory Algorithm to Support En Route and Terminal Area Self-Spacing Concepts*, NASA CR-2007-214899, 2007.

32. Abbott, T. S.; and Swieringa , K. A.: *An Overview of a Trajectory-Based Solution for En Route and Terminal Area Self-Spacing: Eighth Revision*, NASA TM-2017-219662, 2017.

33. Abbott, T. S.: *A Trajectory Algorithm to Support En Route and Terminal Area Self-Spacing Concepts: Fourth Revision*, NASA CR-2018-219828, 2018.

34. RTCA: Minimum Operational Performance Standards (MOPS) for Flight-deck Interval Management (FIM), DO-361, 2015.

35. Olson, Wayne M.: *Aircraft Performance Flight Testing*, AFFTC-TIH-99-01, 2000.